Numerical methods for unconstrained minimization : an integrated computational environment

Kari Miettinen

22 February 1993

URN:NBN:fi-fe19991249 (PDF version)

Helsingin yliopiston verkkojulkaisut
Helsinki 1999

Contents

List of Figures

1 Introduction

This study discusses the methods, algorithms and implementation techniques involved in the computational solution of unconstrained minimization of multidimensional functions. The main goal in this study is to implement a computational environment, that is, a minimization software package, for running and testing a few commonly used methods for unconstrained minimization. This approach divides this study in three parts – *theory* , *implementation* and *testing* , as follows :

- **Minimization strategies : theory and algorithms (chapter 2)**
  discusses the theoretical backround of the minimization algorithms to be implemented.
- **Design and implementation of the minimization software package (chapter 3)**
  introduces the overall design of the minimization software and in greater detail describes the individual software modules, which, as a whole, implement the software package. This chapter also provides instructions for installing and using the software.
- **Testing of the minimization algorithms (chapter 4)**
  introduces the techniques for testing the minimization algorithms, describes the set of test problems used, and discusses the test results.

This chapter is introductory and provides an overview of the general unconstrained minimization problem and explains the backround and motivation for doing this study. Appendices in the end of this paper contain the complete source code of the minimization routines, including the definitions of the available mathematical functions.

1.1 Overview of solving the general unconstrained minimization problem with computational methods

A general unconstrained minimization problem is :

$$\begin{aligned} Given \quad & f : R^n \longrightarrow R \\ find \quad & \mathbf{x}_* \in R^n \; for \; which \; f(\mathbf{x}_*) \leq f(\mathbf{x}) \\ for \quad & every \;\; \mathbf{x} \in R^n \end{aligned} \tag{1}$$

where R^n denotes the n-dimensional Euclidean space. Usually (1) is abbreviated to :

$$\min_{\mathbf{x} \in R^n} \; f : R^n \longrightarrow R \tag{2}$$

The function f is often called the *object* function. In what follows we assume usually that f has continuous second order partial derivatives although sometimes weaker assumptions might suffice.

A minimum point for f can be either *global* i.e. truly the lowest function value or *local* i.e. the lowest function value in a finite neighborhood and not on the boundary of that neighborhood. Virtually nothing is known, however, about finding computationally the absolute lowest point (global minimum) of f in the case when there are many distinct local minimizers, solutions to (1) in open connected regions of R^n. This problem has not been extensively studied in general and is not addressed in this study.

A *constrained* version of (2) would be :

$$\min_{\mathbf{x}\in\Omega\subset R^n} f : R^n \longrightarrow R \tag{3}$$

where Ω is a closed connected region. Again, less is known about how the constrained problem should be solved by computational methods and the topic is not discussed in this study.

A wide variety of algorithms solving unconstrained minimization problems can be found in literature – usually implemented in some computer language (e.g. Fortran or C) and often available in machine-readable format. Powerful numerical libraries (such as NAG) in larger computer systems also provide routines for solving these problems. These methods are usually said to be "global" or "globally convergent" to denote a method that is designed to converge to a *local* minimizer of a function from "almost any starting point". It might be appropriate to call these methods *local* or *locally convergent* , but these descriptions are already reserved by tradition for another usage.

There are two standard heuristics that nearly all of these methods use : (i) find local minimum starting from widely varying starting values of the independent variables, and then pick the lowest of these, or (ii) perturb a local minimum by taking a finite step away from it, and then see if the routine returns a better point or "always" the same one.

1.2 The motivation and purpose of this study

An ordinary user who has found an algorithm or routine (either from literature or from any numerical library) for minimizing his particular function, should without exception have fairly extensive programming skills in order to use it. Normally, any usage of these algorithms or routines requires creating a driver program of some kind or modifications to the ready-made driver programs – if any. Then the user must supply a subroutine to evaluate the object function(s) and a starting point $\mathbf{x}_0$ (a vector in the domain of definition of the object function) that should be a crude approximation to the solution $\mathbf{x}_*$. Some methods require also a subroutine for gradient evaluation. In other words : for each different minimization problem the source code or routine's parameters must be modified manually.

The main goal in this study is to implement an efficient and easy-to-use software package running in personal computers for minimization of multidimensional functions. This package will provide an user-interface for various minimization routines which can solve unconstrained problems of dimensions 1 to 3. Six different minimization methods will be made available :

- Parabolic interpolation (Brent's method) in one-dimension
- Downhill simplex method in multidimensions by Nelder and Mead
- Direction set (Powell's) method in multidimensions
- Variable metric method in multidimensions by Fletcher and Nash
- Conjugate gradients method in multidimensions by Fletcher,Reeves and Nash
- Modified steepest descent method in multidimensions by Vuorinen and the author of this study

By using the interface for these minimization routines the user should be able to do the following :

- Choose the dimension n for the function $f : R^n \longrightarrow R$ to be minimized, $n = 1, 2, 3$.
- Enter the definition of desired function f either manually or using a ready-made file containing this definition. Manually entered definition should be of format

  1. $f(x)$ = expression of x
  2. $f(x, y)$ = expression of x and y
  3. $f(x, y, z)$ = expression of x, y and z

  Functions of other (e.g. parametric) forms should be supplied in ready-made external files.
- Choose the minimization method from the set described above.
- Choose the starting point $\mathbf{x}_0$ (a vector in the domain of definition of function f)
- In one-dimensional case view the function graph before minimization and possibly modify the starting point(s).

When the user has entered all the initial information, the search for a minimum is started. If the minimum is found, it is displayed with a 2- or 3-dimensional plot of function, whenever plotting is possible. The user has also possibility of obtaining information of how the method converged to a minimum i.e. which was the minimization route, and how much computational time was spent. If the minimization fails for some reason, some "other reliable software" (included in this computational environment) should be used. By using the interface described above, the user should also be able to evaluate the performance of each minimization method without any function- or route-plotting, and verify how the results compare with this "other software" that solves the same problem.

This "other software" is chosen to be $Mathematica^{TM}$ which is a general software system for mathematical and other applications, including a minimization routine suitable for this purpose. The minimization routine in *Mathematica* works by following the

steepest descent from each point it reaches. The other minimization algorithms used are implemented in ANSI-standard C programming language. *Mathematica* in addition, has an important role in implementing the minimization software : it is not just a tool providing a realiable routine for comparing the results of other routines, but a sole heart of the whole environment. The design and implementation of the minimization software and the role of *Mathematica* is fully described in chapter 3.

2 Minimization strategies : theory and algorithms

This chapter discusses the theoretical backround of the algorithms to be implemented in the minimization software package. References to the source code, included in the appendices, are made frequently.

2.1 Bracketing a minimum of a one-dimensional function

A *root* of a continuous function $f : R \longrightarrow R$ is known to be bracketed by a pair of points, $a < b$, $a, b \in R$ when the function has opposite sign at those two points i.e. $f(a)f(b) < 0$. A *minimum* , by contrast, is known to be bracketed only when there is a *triplet* of points

$$a < b < c \ , \ a, b, c \in R \ , \tag{4}$$

such that

$$f(b) < f(a) \ and \ f(b) < f(c). \tag{5}$$

In this case we know that the function has a minimum in the interval (a, c).

Initial bracketing of a minimum is usually considered as an essential part of any one-dimensional minimization and it is also used by minimization methods described in the next sections. After the initial bracketing triplet is found, a simple minimization algorithm would continue the process as follows :
Given triplet of points $a < b < c$, choose a new point u , either in the interval (a, b) or (b, c). Suppose, that we make the latter choice : if $f(b) < f(u)$ then the new bracketing triplet of points is $a < b < u$; contrariwise, if $f(b) > f(u)$, then the new bracketing triplet is $b < u < c$. In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far. This process of bracketing is continued until the distance between the two outer points of the triplet is tolerably small.

2.1.1 Implemented routines : MNBRAKN and MNBRAK1

Routines in appendices A and B have both the same name *mnbrak*, but they are called here with names of their modules : *Mnbrak1* and *Mnbrakn*, respectively.

Routine *Mnbrakn* (in Appendix B) by Press et al. [Press 1986] implements the initial bracketing of a minimum for a given one-dimensional function using the following strategy :
Given a function $f : R \longrightarrow R$ and given distinct initial points $a < b$, $a, b \in R$, search

for a bracketing triplet in the downhill direction (defined by the function evaluated at the initial points), starting with some initial guess for point c and then increasing the step size at each step by a constant factor (defined by golden ratio) , or else by the result of a parabolic extrapolation of the preceding points that is designed to lead to the extrapolated turning point. Return the points $a < b < c$ which bracket the minimum.

Another bracketing routine is also implemented here, namely *Mnbrak1* (in Appendix A) by author of this study. *Mnbrak1* uses the following strategy :
Given a function $f : R \longrightarrow R$ and given distinct initial points $a_0 < b_0$, $a_0, b_0 \in R$, search for a bracketing triplet first in the downhill direction with step size defined by golden ratio, as with routine *Mnbrakn* above (without parabolic extrapolation, though), and if this was not successful, try uphill direction. If there was still no success, try both directions with larger step size. In each step, the interval (a_i, b_i) , $i = 1, 2, \ldots$ is searched for the third point c of the bracketing triplet by dividing it into smaller intervals of equal size and testing the endpoints of each interval. The first guess for c is the midpoint $\frac{a_i+b_i}{2}$. The choice of the next interval to be searched,that is (a_{i+1}, b_{i+1}), depends on the direction we want to follow. When following the *downhill direction*, and if a_i is the endpoint having the lowest function value, that is $f(a_i) < f(b_i)$, we would set :

$$\begin{aligned} a_{i+1} &= a_i - k(b_0 - a_0) \\ b_{i+1} &= \frac{a_i + b_i}{2} \end{aligned} \tag{6}$$

where k is a constant. Should b_i have been the lowest point, we would have set :

$$\begin{aligned} a_{i+1} &= \frac{a_i + b_i}{2} \\ b_{i+1} &= b_i + k(b_0 - a_0) \end{aligned} \tag{7}$$

When following the *uphill direction*, the next interval is achieved by using equation (6) when $f(b_i) < f(a_i)$ and (7) when $f(a_i) < f(b_i)$.
The constant k is set to 1.618034 (golden ratio) during the first major iteration loop and to 50.0 during the second. Note, that during this algorithm the role of points b and c in (4) is reversed, that is $a < c < b$. Points b and c are switched in the end of the routine, thus returning the points $a < b < c$ which bracket the minimum.

The reason for implementing an alternative bracketing algorithm was the fact that routine *Mnbrakn* does not work with some simple functions. *Mnbrakn* failes e.g. with cubic function $f(x) = x^3 - 2x + 5$ (which has a local minimum $f(x) \simeq 3.911$ at $x = \sqrt{\frac{2}{3}}$), when using e.g. $a = -2$, $b = 1$ as the initial points. Because *Mnbrakn* always follows the downhill direction increasing the step size, and in this case $\lim_{x \to -\infty} f(x) = -\infty$, it fails in bracketing by exceeding the floating-point range. Routine *Mnbrak1* has many weaknesses, though, sometimes requiring hundreds of iterations to find the bracketing triplet. It is recommended that the usage of *Mnbrak1* is limited only for one-dimensional problems because of its time-consuming feature (and not to be used as a subalgorithm for multidimensional methods described in the later sections).

2.2 Parabolic interpolation and Brent's method in one-dimension

The idea in Brent's method [Brent 1973] for minimizing a function $f : R \longrightarrow R$, is to use function values and their associated points to generate some simple function, call it $I(x)$, which *interpolates* function $f(x)$ between the points at which f has been computed. The most popular choices for $I(x)$ are polynomials of degree 2 (parabolic approximation) or degree 3 (cubic approximation). Brent uses parabolic approximation and makes an assumption that if the function is nicely parabolic near to the minimum then the parabola fitted through any known three points x_0, x_1, x_2 should lead to the minimum in a single leap.

Consider the case where $f(x)$ is known at three points $x_0 < x_1 < x_2$. Then using a parabola to interpolate the function requires

$$f(x_j) = A + Bx_j + Cx_j^2 \ , \ j = 0, 1, 2 \tag{8}$$

which provides three linear equations for three unknowns A,B and C. Once these are found, it is a simple task to set the derivative of the interpolant to zero

$$\frac{dI(x)}{dx} = 2Cx + B = 0 \tag{9}$$

to find a value of x which minimizes the interpolating polynomial $I(x)$.
Consider these three points $x_0 < x_1 < x_2$, where, using the notation

$$f_j = f(x_j) \tag{10}$$

for brevity,

$$f_1 \leq f_0 \tag{11}$$

and

$$f_1 \leq f_2 \ . \tag{12}$$

Excluding the exceptional cases that the function is flat or otherwise perverse, so that at least one of the conditions (11) or (12) is a strict inequality, the interpolating parabola will have its minimum between x_0 and x_2. Now all the distances from x_1 can be measured, so that equations (8) can be rewritten

$$\begin{aligned} f_0 &= A + Bd_0 + Cd_0^2 \\ f_1 &= A \\ f_2 &= A + Bd_2 + Cd_2^2 \end{aligned} \tag{13}$$

where

$$d_j = x_j - x_1 \ , \ j = 0, 1, 2 \tag{14}$$

Equations (13) can be solved by elimination to give

$$B = \frac{(f_0 - f_1)d_2^2 - (f_2 - f_1)d_0^2}{d_0 d_2^2 - d_2 d_0^2} \tag{15}$$

and

$$C = \frac{(f_0 - f_1)d_2 - (f_2 - f_1)d_0}{d_0^2 d_2 - d_2^2 d_0} \tag{16}$$

Hence the minimum of the parabola is found at

$$x = \frac{-B}{2C} = \frac{1}{2}\frac{(f_0 - f_1)d_2^2 - (f_2 - f_1)d_0^2}{(f_0 - f_1)d_2 - (f_2 - f_1)d_0} \tag{17}$$

Since the abscissa rather than the ordinate is wanted to be found, this approach is technically called *inverse parabolic interpolation*.

2.2.1 Implemented routines : BRENT

Routine *Brent* (in Appendix C), adapted from Press et al [Press 1986], implements the minimizing of given one-dimensional function using Brent's method :
Given a function $f : R \longrightarrow R$ and given a bracketing triplet of abscissas $a < b < c$, $a, b, c \in R$ which satisfy (5), isolate the minimum to a fractional precision of value TOL, (usually a a square root of machine's floating-point precision). At any particular stage keep track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows : the minimum is bracketed between a and b ; x is the point with the very least function value found so far; w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Try parabolic fit through points x, v, w : to be acceptable, the parabolic step must (i) fall within the bounding interval (a, b), and (ii) imply a movement from the best current value x that is *less* than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually converging to something. If the parabolic steps are of no use, the routine will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. Return the abscissa x of the minimum (x is the midpoint of points a, b) when it is fractionally accurate enough.

2.3 Downhill simplex method in multidimensions

This section begins the consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variables. In one-dimensional minimization it was possible to bracket a minimum, so that the success of a subsequent isolation was guaranteed. There is no analogous procedure in multidimensional space, though. For multidimensional minimization, the best one can do is give algorithm a starting guess, that is, an n-vector of independent variables as the first point to try. The algorithm is then supposed to go downhill through the n-dimensional topography, until it encounters an (at least local) minimum.

The first of these methods, Downhill simplex method, is a search procedure based on heuristic ideas. Its strengths are that it requires no derivatives to be computed, so it can cope with functions which are not easily written as analytic expressions, and that it always increases the information available concerning the function by reporting its value

at a number of points. Its weakness is primarily that it does not use this information very effectively, and so may take an unnecessarily large number of function evaluations to locate a solution. This method is due to Nelder and Mead [Nelder-Mead 1965].

A *simplex* is a geometrical figure consisting, in n dimensions, of $n+1$ points (vertices) and all their interconnecting line segments and polygonal faces. In one-dimension, a simplex is just a line segment, in two dimensions it is a triangle and in three dimensions a tetrahedron. Simplexes used in this method ought to be nondegenerate, i.e. enclose a finite and positive inner n-dimensional volume. If any vertex of a nondegenerate simplex is taken as the origin, then the n other vertices define vector directions that span the n-dimensional vector space.

The downhill simplex method must be started with $n+1$ points, defining an initial simplex. Consider one of these points as being an initial starting point $\mathbf{p}_0$ in the n-dimensional Euclidean space. A common choice for other n points would be then

$$\mathbf{p}_i = \mathbf{p}_0 + \lambda \mathbf{e}_i \ , \ i = 1, \ldots, n \tag{18}$$

where each $\mathbf{e}_i$ is an unit vector, and where λ is a constant approximating the problem's characteristic length scale.

The essence of the downhill simplex method is as follows : the function $f : R^n \longrightarrow R$ is evaluated at each point (vertex) of the simplex and the vertex having the highest function value is replaced by a new point with a lower function value. This is done in such a way that "the simplex adapts itself to the local landscape and, contracts on to the final minimum." There are three main operations which are made on the simplex : *reflection,expansion and contraction*. In order to operate on the simplex, it is necessary to order the vertices so that the highest is $\mathbf{p}_H$, the next-to-highest $\mathbf{p}_N$, and the lowest $\mathbf{p}_L$. Thus the associated function values obey

$$f(\mathbf{p}_H) \geq f(\mathbf{p}_N) \geq f(\mathbf{p}_i) \geq f(\mathbf{p}_L) \tag{19}$$

for vertices $\mathbf{p}_i$ where $i \neq H, N or L$. Method takes a series of steps, most steps just moving the vertex $\mathbf{p}_H$ through the opposite face of the simplex to a lower point. These steps are called *reflections*, and they are constructed to conserve the volume of the simplex (hence maintain its nondegeneracy). When the reflection step is successful (i.e. a new point having a lower function value than $\mathbf{p}_L$ is found), the method *expands* the simplex in the direction of reflection to take larger steps. If the reflection step fails (i.e. the reflected point has higher function value than $\mathbf{p}_N$) the simplex is *contracted* in the tranverse direction. If the contraction step gives no improvement, the method contracts the simplex in all directions by moving the vertices half-way toward the current best point. An appropriate sequence of such steps will always converge to a minimum of the function. Termination criteria for this method is that the decrease in the function value is fractionally smaller than some predefined tolerance value.

2.3.1 Implemented routines : AMOEBA

Routine *Amoeba* (in Appendix D), adapted from Press et al [Press 1986], implements the minimizing of given n-dimensional function using downhill simplex method :

Given a function $f : R^n \longrightarrow R$ and given a starting simplex of $n+1$ vertices (i.e. $n(n+1)$ matrix $\mathbf{P}$) and values of f evaluated at the vertices (i.e. $n+1$ -vector $\mathbf{y}$), find the minimum using downhill simplex method. Return the vertices of a simplex minimizing f so that function values at these vertices are all within pre-defined tolerance $FTOL$ of a minimum function value.

2.4 Direction set (Powell's) method in multidimensions

Section 2.2 provided a method for minimizing a function of one variable. This method can also be used in multidimensional cases as follows : if we start at a point $\mathbf{p}$ in n-dimensional space and proceed from there in some vector direction $\mathbf{u}$, then any function f of n variables can be minimized along the line $\mathbf{u}$ by using a one-dimensional method.

Many algorithms using this approach can be found in literature and they consists of sequences of such *line minimizations*. Different methods will differ only by how, at each stage, they choose the next direction $\mathbf{u}$ to try. All such methods presume the existence of a subalgorithm, call it *Linmin* , whose definition can now be taken as :
LINMIN: Given as input vectors $\mathbf{p}$ *and* $\mathbf{u}$ *and the function* f*, find the scalar* λ *that minimizes* $f(\mathbf{p} + \lambda\mathbf{u})$. *Replace* $\mathbf{p}$ *by* $\mathbf{p} + \lambda\mathbf{u}$ *, and replace* $\mathbf{u}$ *by* $\lambda\mathbf{u}$.

The point of the line $\{ \mathbf{p} + t\mathbf{u} : t \in R \}$ where the minimum is attained is called the *line minimum (point)* of the function.

Consider the case, where explicit computation of the function's gradient is not possible and thus unusable in the choice of directions – then the simplest method would be this: Take the unit vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$ as a *set of directions*. Using *Linmin*, move along the first direction to its minimum, then from there along the second direction to its minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

This simple method is, however, very inefficient for some functions. Consider a function of two dimensions whose contour map defines a long, narrow "valley" at some angle to the coordinate basis vectors. Then the only way down to the minimum going along the basis vectors at each stage is by a series of many tiny steps, crossing and re-crossing the principal axis.

Obviously better set of directions than the unit vectors, is needed. All *direction set methods* consists of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along other, so that interminable cycling through the set of directions can be avoided.

2.4.1 Conjugate directions

The concept of "non-interfering" directions, more conventionally called *conjugate directions* is now made mathematically explicit.

First, note that if we minimize a function along some direction $\mathbf{u}$ then the gradient of the function must be perpendicular to $\mathbf{u}$ at the line minimum; if not, then there would still be a nonzero directional derivative along $\mathbf{u}$. Next take some particular point $\mathbf{p}$ in n-dimensional space as the origin of the coordinate system with coordinates $\mathbf{x}$. Then any function f having continuous second order partial derivatives can be approximated by its Taylor series

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{p}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \ldots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{H} \cdot \mathbf{x} \end{aligned} \tag{20}$$

where

$$c \equiv f(\mathbf{p}) \quad \mathbf{b} \equiv -\nabla f \mid \mathbf{p} \quad [\mathbf{H}]_{ij} \equiv \frac{\partial^2 f}{\partial x_i \partial x_j} \mid \mathbf{p} \tag{21}$$

The matrix $\mathbf{H}$ whose components are the second partial derivative matrix of the function is called the $Hessian\ matrix$ of the function at $\mathbf{p}$. In the approximation of (20), the gradient of f is easily calculated as

$$\nabla f = \mathbf{H} \cdot \mathbf{x} - \mathbf{b} \tag{22}$$

which implies that the gradient will vanish at a value of $\mathbf{x}$ obtained by solving $\mathbf{H} \cdot \mathbf{x} = \mathbf{b}$.

Consider then the change of the gradient ∇f when moving along some direction. Evidently

$$\partial(\nabla f) = \mathbf{H} \cdot (\partial \mathbf{x}). \tag{23}$$

Suppose, that we have moved along some direction $\mathbf{u}$ to a minimum and now propose to move along some new direction $\mathbf{v}$. The condition that motion along $\mathbf{v}$ does not "spoil" the minimization along $\mathbf{u}$ is that the gradient stay perpendicular to $\mathbf{u}$, i.e. that the change in the gradient be perpendicular to $\mathbf{u}$. By equation (23) this is just

$$0 = \mathbf{u} \cdot \partial(\nabla f) = \mathbf{u} \cdot \mathbf{H} \cdot \mathbf{v} \tag{24}$$

When (24) holds for two vectors $\mathbf{u}$ and $\mathbf{v}$, they are said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a *conjugate set.* If successive line minimizations of a function are made along a conjugate set of directions, then none of these directions have to be tried again. The best possible situation for a direction set method is to come up with a set of n linearly independent, mutually conjugate directions. Then, one pass of n line minimizations will lead exactly at the minimum of a quadratic form like (20). For functions which are not exactly quadratic forms, this won't be the case, but repeated cycles of n line minimizations will in due course converge *quadratically* to the minimum.

2.4.2 Powell's quadratically convergent method

Powell, in 1964, discovered a direction set method which does produce n mutually conjugate directions using the following strategy :
Initialize the set of directions $\mathbf{u}_i$ to the basis vectors,

$$\mathbf{u}_i = \mathbf{e}_i \ , \ i = 1, \ldots, n \tag{25}$$

and then repeat the following sequence of steps until the function stops decreasing :

- Save the starting position as $\mathbf{p}_0$.
- For $i = 1, \ldots, n$, move $\mathbf{p}_{i-1}$ to the minimum along direction $\mathbf{u}_i$ and call this point $\mathbf{p}_i$.
- For $i = 1, \ldots, n-1$, set $\mathbf{u}_i \longleftarrow \mathbf{u}_{i+1}$.
- Set $\mathbf{u}_n \longleftarrow \mathbf{p}_n - \mathbf{p}_0$.
- Move $\mathbf{p}_n$ to the minimum along direction $\mathbf{u}_n$ and call this point $\mathbf{p}_0$.

Powell showed that, for quadratic form like (20), k iterations of the above basic procedure produce a set of directions $\mathbf{u}_i$ whose last k members are mutually conjugate. Therefore, n iterations of this procedure, amounting $n(n+1)$ line minimizations in all, will exactly minimize a quadratic form.

2.4.3 Implemented routines : POWELL and LINMIN

Routine *Powell* (in Appendix G), adapted from Press et al [Press 1986], implements the minimizing of given n-dimensional function using method described above with exception that the property of quadratic convergence is somewhat discarded in favor of a more heuristic scheme (a version of Powell's method in Acton [Acton 1970]) which tries to find a few proper directions along narrow valleys instead of n necessary conjugate directions. This includes (i) discarding the direction along which the function made its largest decrease in order to avoid building up linear dependence and (ii) discarding adding of the new direction $\mathbf{p}_n - \mathbf{p}_0$ in some cases.
Given a function $f : R^n \longrightarrow R$ and given an n-dimensional starting point $\mathbf{p}$ and the initial set of directions $\mathbf{u}_i$ routine *Powell* then finds the minimum. When the routine has failed in decreasing the function value by more than pre-defined tolerance-value $FTOL$ in one iteration, it returns the best point found and the function value at this point.

Routine *Powell* uses subroutine *Linmin* (in Appendix F), adapted from Press et al [Press 1986], for line minimizations. Routine *Linmin* works as follows :
Given an n-dimensional point $\mathbf{p}$ and direction $\mathbf{u}$ move and reset $\mathbf{p}$ to the point where the function f takes on a minimum along the direction $\mathbf{u}$ from $\mathbf{p}$ and replace $\mathbf{u}$ by the actual vector displacement that $\mathbf{p}$ was moved. Routine *Linmin* returns the value of f at the returned location $\mathbf{p}$. This is actually all accomplished by calling the routines *Mnbrakn* and *Brent* (see section 2.1.1 why routine *Mnbrakn* is used instead of *Mnbrak1*).

2.5 Overview of the descent methods for multidimensional minimization

In the next sections methods will be considered which make use of the gradient of the function $f(\mathbf{x})$ which will now be called $\mathbf{g}$:

$$g_j = \frac{\partial f(\mathbf{x})}{\partial x_j} \ , \ for \ j = 1, 2, \ldots, n \tag{26}$$

evaluated at the point $\mathbf{x}$. So called *descent methods* all use the basic iterative step

$$\mathbf{x}' = \mathbf{x} - k\mathbf{B} \cdot \mathbf{g} \tag{27}$$

where $\mathbf{B}$ is a matrix defining a transformation of the gradient and k is a step length. The simplest such algorithm, the method of *steepest descents* , was proposed by Cauchy, in 1848, for the solution of systems of nonlinear equations. This uses

$$\mathbf{B} = \mathbf{1}_n \tag{28}$$

and any step length k which reduces the function so that

$$f(\mathbf{x}') < f(\mathbf{x}) \tag{29}$$

The definition of *steepest descent method*, can now be taken as :
STEEPEST DESCENT: Start at point $\mathbf{x}_0 \in R^n$. *As many times as needed, move from point* $\mathbf{x}_i$ *to the point* $\mathbf{x}_{i+1}$ *by minimizing along the line from* $\mathbf{x}_i$ *in the direction of the local downhill gradient* $-\mathbf{g}(\mathbf{x}_i)$.

The principal difficulty with steepest descents method is that the search directions generated are not linearly independent [Kowalik-Osborn 1968]. When minimizing e.g. a function of two dimensions whose contour map defines a long, narrow valley, method tends to criss-cross the valley instead of following the valley floor to the minimum. Thus a number of methods have been developed which aim to transform the gradient $\mathbf{g}$ so that the search directions generated in (27) are linearly independent or, equivalently, are *conjugate* to each other with respect to some positive definite matrix $\mathbf{A}$. Recall, that if $\mathbf{u}$ and $\mathbf{v}$ are search directions, they are conjugate with respect to the positive definite matrix $\mathbf{A}$ if

$$\mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} = 0 \tag{30}$$

A modified version of steepest descent algorithm is described in section 2.8. The conjugate gradients method (described in section 2.7) generates such a set of search directions implicitly. The variable metric method (described in section 2.6) uses a tranformation matrix which is adjusted at each step to generate approppriate search directions. There is, however, another way to think of this process. Consider the set of nonlinear equations formed by the gradient at a minimum

$$\mathbf{g}(\mathbf{x}') = 0. \tag{31}$$

It is possible to seek such solutions via a linear approximation from the current point **x**, that is

$$\mathbf{g}(\mathbf{x}') = \mathbf{g}(\mathbf{x}) + \mathbf{H}(\mathbf{x}) \cdot (\mathbf{x}' - \mathbf{x}) \tag{32}$$

where **H**(**x**) is the Hessian matrix

$$[\mathbf{H}]_{ij} \equiv \frac{\partial g_i(\mathbf{x})}{\partial x_j} \equiv \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \tag{33}$$

of second derivatives of the function to be minimized or first derivatives of the nonlinear equations. For the current purpose, suppose that $\mathbf{H}$ is positive definite so that its inverse exists. Using the inverse together with equations (31) implies

$$\mathbf{x}' = \mathbf{x} - \mathbf{H}^{-1}(\mathbf{x}) \cdot \mathbf{g}(\mathbf{x}) \tag{34}$$

which is Newton's method in n parameters. This is equivalent to equation (27) with

$$\mathbf{B} = \mathbf{H}^{-1} \quad k = 1. \tag{35}$$

The step parameter k is rarely fixed, however, and usually some form of linear search is used (such as *Linmin*).

Using Newton's method requires n^2 second derivative evaluations, n first derivative evaluations and a matrix inverse even before the linear search could be attempted. Evaluations can be very expensive, and for this reason Newton's method does not recommend itself for most problems.

2.6 Descent to a minimum : Variable metric methods in multidimensions

Suppose now, that a method could be found to approximate $\mathbf{H^{-1}}$ directly from the first derivative information available at each step of the iteration defined by (27). This would save a great deal of work in computing both the derivative matrix $\mathbf{H}$ and its inverse. This is precisely the role of the matrix $\mathbf{B}$ in generating the conjugate search directions of the variable metric family of algorithms, and has led their being known also as *quasi-Newton* methods.

2.6.1 Variable metric algorithms

Variable metric algorithms have proved to be the most effective class of general-purpose methods for solving unconstrained minimization problems. All the variable metric methods seek to minimize the function $f(\mathbf{x})$ of n variables by means of sequence of steps (27) :

$$\mathbf{x}' = \mathbf{x} - k\mathbf{B} \cdot \mathbf{g}$$

The definition of an algorithm of this type consists in specifying (i) how the matrix $\mathbf{B}$ is computed, and (ii) how k is chosen (a line minimization problem). Because the second

choice can be made using previous methods (Linmin) or with simpler search procedures, the rest of this section is devoted to the task of deciding appropriate conditions on the matrix $\mathbf{B}$.

Firstly, when it works, Newton's method generally converges very rapidly. Thus it would seem desirable that $\mathbf{B}$ tend in some way towards the inverse Hessian matrix $\mathbf{H^{-1}}$. However, the computational requirement in Newton's method for second partial derivatives and for solution of linear-equation systems at each iteration must be avoided. Secondly, $\mathbf{B}$ should be positive definite to permit the algorithm to exhibit quadratic termination, that is, to minimize a quadratic form in at most n steps (27). A quadratic form (20) has a unique minimum if $\mathbf{H}$ is positive definite.

If n linearly independent directions $\mathbf{t}_j$, $j = 1, \ldots, n$ exists for which

$$\mathbf{B} \cdot \mathbf{H} \cdot \mathbf{t}_j = \mathbf{t}_j \tag{36}$$

then

$$\mathbf{B} \cdot \mathbf{H} = \mathbf{1}_n \quad or \quad \mathbf{B} = \mathbf{H}^{-1}. \tag{37}$$

The search directions $\mathbf{t}_j$ are sought conjugate to $\mathbf{H}$, leading in an implicit fashion to the expansion

$$\mathbf{H}^{-1} = \sum_{j=1}^{n} \frac{\mathbf{t}_j \cdot \mathbf{t}_j}{\mathbf{t}_j \cdot \mathbf{H} \cdot \mathbf{t}_j} \tag{38}$$

Since $\mathbf{B}$ is to be developed as a sequence of matrices $\mathbf{B}_m$, the condition (36) will be stated

$$\mathbf{B}_m \cdot \mathbf{H} \cdot \mathbf{t}_j = \mathbf{t}_j. \tag{39}$$

Thus we have

$$\mathbf{B}_{n+1} = \mathbf{H}^{-1}. \tag{40}$$

For a quadratic form, the change in the gradient at $\mathbf{x}_j$, that is

$$\mathbf{g}_j = \mathbf{H} \cdot \mathbf{x}_j - \mathbf{b} \tag{41}$$

is given by

$$\begin{aligned} \mathbf{y}_j &= \mathbf{g}_{j+1} - \mathbf{g}_j \\ &= \mathbf{H} \cdot (\mathbf{x}_{j+1} - \mathbf{x}_j) \\ &= k_j \mathbf{H} \cdot \mathbf{t}_j \end{aligned} \tag{42}$$

since the elements of $\mathbf{H}$ are constant in this case. From this it follows that (39) becomes

$$\mathbf{B}_m \cdot \mathbf{y}_j = k_j \mathbf{t}_j. \tag{43}$$

Assuming (39) is correct for $j < m$, a new step

$$\mathbf{t}_m = \mathbf{B}_m \cdot \mathbf{g}_m \tag{44}$$

is required to be conjugate to all previous directions $\mathbf{t}_j$, i.e.

$$\mathbf{t}_j \cdot \mathbf{H} \cdot \mathbf{t}_m = 0 \quad for\ j < m. \tag{45}$$

But from equations (42),(43) and (44) we obtain

$$\begin{aligned} \mathbf{t}_j \cdot \mathbf{H} \cdot \mathbf{t}_m &= \frac{\mathbf{y}_j \cdot \mathbf{t}_m}{k_j} \\ &= \frac{\mathbf{y}_j \cdot \mathbf{B}_m \cdot \mathbf{g}_m}{k_j} \\ &= \mathbf{t}_j \cdot \mathbf{g}_m \end{aligned} \tag{46}$$

Suppose now that the linear searches at each step have been performed accurately. Then the directions

$$\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_{m-1} \tag{47}$$

define a hyperplane on which the quadratic form has been minimized. Thus $\mathbf{g}_m$ is orthogonal to $\mathbf{t}_j$, $j < m$, and (45) is satisfied, so that the new direction $\mathbf{t}_m$ is conjugate to the previous ones.

In order that $\mathbf{B}_{m+1}$ satisfies the above theory so that the process can continue, the update $\mathbf{C}$ in

$$\mathbf{B}_{m+1} = \mathbf{B}_m + \mathbf{C} \tag{48}$$

must satisfy (43). From this, it follows m conditions on the order-n matrix $\mathbf{C}$. This degree of choice in $\mathbf{C}$ has in part been responsible for the large literature on variable metric methods and some references are given in the next section how this correction term should be constructed.

The essence of the variable metric methods, i.e. that information regarding the Hessian has been drawn from first derivative computations only, can be found in the above development. The variable metric methods generate an approximate inverse Hessian as they proceed, requiring only one evaluation of the first derivatives per step and reduce the function value at each of these steps.

2.6.2 Implemented routines : VMMIN

Routine *Vmmin* (in Appendix H), adapted from Nash [Nash 1990], implements the minimizing of given n-dimensional function using variable metric method by Fletcher [Fletcher 1970] with some modifications done by Nash. It uses an "acceptable point" procedure in performing the linear search. This procedure takes the first point in some generated sequence which satisfies a predefined acceptance criterion. Suppose, that the step taken is

$$\mathbf{t} = \mathbf{x}' - \mathbf{x} = -k\mathbf{B} \cdot \mathbf{g} \tag{49}$$

with $k = 1$ initially. The decrease in the function

$$\Delta f = f(\mathbf{x}') - f(\mathbf{x}) < 0 \tag{50}$$

will be approximated for small steps $\mathbf{t}$ by the first term in the Taylor series for f along $\mathbf{t}$ from $\mathbf{x}$, that is, by $\mathbf{t} \cdot \mathbf{g}$. This is negative when $\mathbf{t}$ is a downhill direction. By choosing $k = 1, w, w^2, \ldots$, for $0 < w < 1$ succesively, it is always possible to produce a $\mathbf{t}$ such that

$$0 < tolerance < \frac{\Delta f}{\mathbf{t} \cdot \mathbf{g}} \quad for \quad tolerance \quad \ll 1 \tag{51}$$

unless the minimum has been found. This presumes

$$\mathbf{t} \cdot \mathbf{g} < 0 \tag{52}$$

to ensure a downhill direction.

Routine *Vmmin* uses $tolerance = 0.0001$ and $w = 0.2$ as predefined values. The method used for building up the update matrix $\mathbf{C}$ in (48) goes by the name Broyden-Fletcher-Shanno from Broyden [Broyden 1970], Fletcher [Fletcher 1970] and Shanno [Shanno 1970]. The matrix $\mathbf{B}$ is initially set to the unit matrix ($\mathbf{1}_n$) and may be reset to unity again in some cases to restart the searching.
Given a function $f : R^n \longrightarrow R$ with its first partial derivatives and given an n-dimensional starting vector $\mathbf{xvec}$, routine *Vmmin* then finds the minimum. The algorithm is taken to have converged if either of the following cases occur during the first step after matrix $\mathbf{B}$ is reset to unity : (i) direction of search is uphill i.e. $\mathbf{t} \cdot \mathbf{g} \geq 0$ (ii) no change is made in the parameters by the linear search along $\mathbf{t}$, i.e. $\mathbf{x}' = \mathbf{x}$. Routine returns the location of the minimum $\mathbf{x}$ and also the minimum value of the function $fmin$.

2.7 Descent to a minimum : Conjugate gradients methods in multidimensions

The variable metric method requires working space proportional to n^2, where n is the number of parameters in the function to be minimized. The parameters $\mathbf{xvec}$ and gradient $\mathbf{g}$ require only n elements each, so it is worth considering algorithms which make use of this information without the requirement that it be collected in a matrix. This approach leads us to the principle ideas of conjugate gradients methods.

2.7.1 Conjugate gradients algorithm

Consider the quadratic form (20)

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x} \cdot \mathbf{H} \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x} + c$$

of which the gradient at $\mathbf{x}$ is (41)

$$\mathbf{g} = \mathbf{H} \cdot \mathbf{x} - \mathbf{b}$$

Then if the search direction at some iteration j is $\mathbf{t}_j$, we have (42)

$$\begin{aligned} \mathbf{y}_j &= \mathbf{g}_{j+1} - \mathbf{g}_j \\ &= k_j \mathbf{H} \cdot \mathbf{t}_j \end{aligned}$$

where k_j is the step-length parameter.

If any initial step $\mathbf{t}_1$ is made subject only to its being downhill, that is

$$\mathbf{t}_1 \cdot \mathbf{g}_1 < 0 \tag{53}$$

then the construction of search directions $\mathbf{t}_i$, $i = 1, 2, \ldots, n$, conjugate with respect to the Hessian $\mathbf{H}$, is possible via the Gram- Schmidt process. That is to say, given an arbitrary new downhill direction $\mathbf{q}_i$ at step i , it is possible to construct, by choosing coefficients z_{ij} , a direction

$$\mathbf{t}_i = \mathbf{q}_i + \sum_{j=1}^{i-1} z_{ij}\mathbf{t}_j \tag{54}$$

such that

$$\mathbf{t}_i \cdot \mathbf{H} \cdot \mathbf{t}_j = 0 \quad for\ j < i. \tag{55}$$

This is achieved by applying $\mathbf{t}_j \cdot \mathbf{H}$ to both sides of equation (54), giving

$$z_{ij} = -\frac{\mathbf{t}_j \cdot \mathbf{H} \cdot \mathbf{q}_i}{\mathbf{t}_j \cdot \mathbf{H} \cdot \mathbf{t}_j} \tag{56}$$

by substitution of the condition (55) and the assumed conjugacy of the $\mathbf{t}_j$, $j = 1, 2, \ldots, (i-1)$. Note, that the denominator of (56) cannot be zero if $\mathbf{H}$ is positive definite and $\mathbf{t}_j$ is not null.

Now if $\mathbf{q}_i$ is chosen to be negative gradient

$$\mathbf{q}_i = -\mathbf{g}_i \tag{57}$$

and $\mathbf{t}_j \cdot \mathbf{H}$ is substituted from (42), then we have

$$\begin{aligned} z_{ij} &= \frac{\mathbf{g}_i \cdot (\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{t}_j \cdot (\mathbf{g}_{j+1} - \mathbf{g}_j)} \\ &= \frac{\mathbf{g}_i \cdot \mathbf{y}_j}{\mathbf{t}_j \cdot \mathbf{y}_j} \end{aligned} \tag{58}$$

Moreover, if accurate line searches have been performed at each of the $(i-1)$ previous steps, then the function f (of the quadratic form (20)) has been minimized on a hyperplane spanned by the directions $\mathbf{t}_j$, $j = 1, 2, \ldots, (i-1)$, and $\mathbf{g}_i$ is orthogonal to each of these directions. Therefore, we have

$$z_{ij} = 0 \quad for\ j < (i-1) \tag{59}$$

$$z_{i,i-1} = \frac{\mathbf{g}_i \cdot (\mathbf{g}_i - \mathbf{g}_{i-1})}{\mathbf{t}_{i-1} \cdot (\mathbf{g}_i - \mathbf{g}_{i-1})} \tag{60}$$

Alternatively, using

$$\mathbf{t}_{i-1} = -\mathbf{g}_{i-1} + \sum_{j=1}^{i-2} z_{ij}\mathbf{t}_j \tag{61}$$

which is a linear combination of $\mathbf{g}_j$, $j = 1, 2, \ldots, (i-1)$, we obtain

$$z_{i,i-1} = \frac{\mathbf{g}_i \cdot (\mathbf{g}_i - \mathbf{g}_{i-1})}{\mathbf{g}_{i-1} \cdot \mathbf{g}_{i-1}} \tag{62}$$

$$= \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{g}_{i-1} \cdot \mathbf{g}_{i-1}} \tag{63}$$

by virtue of the orthogonality mentioned above.

As in the case of variable metric algorithms, the formulae obtained for quadratic forms are applied in somewhat cavalier fashion to the minimization of general nonlinear functions. The formulae (60),(62) and (63) are now no longer equivalent. For reference, these will be associated with the names : Beale and Sorenson for (60) ; Polak and Ribiere for (62) ; and Fletcher and Reeves [Fletcher-Reeves 1964] for (63).

In summary, the conjugate gradients algorithm proceeds by setting

$$\mathbf{t}_1 = -\mathbf{g}(\mathbf{x}_1) \tag{64}$$

and

$$\mathbf{t}_i = z_{i,i-1}\mathbf{t}_{i-1} - \mathbf{g}_i(\mathbf{x}_i) \tag{65}$$

with

$$\mathbf{x}_{j+1} = \mathbf{x}_j + k_j \mathbf{t}_j \tag{66}$$

where k_j is determined by a linear search for a minimum of $f(\mathbf{x}_j + k_j \mathbf{t}_j)$ with respect to k_j.

2.7.2 Implemented routines : CGMIN

A program to use these ideas requires that a choice be made of (i) a recurrence formula to generate the search directions and (ii) a linear search. Since the conjugate gradients methods are derived on the presumption that they minimize a quadratic form in n steps, but in practical computations this rarely happens, it is necessary to suggest a method for continuing the iterations after n steps.

Routine *Cgmin* (in Appendix I), adapted from Nash [Nash 1990], implements the minimizing of given n-dimensional function using conjugate gradients method by Fletcher and Reeves [Fletcher-Reeves 1964] with some modifications done by Nash. It is restarted every n steps or whenever the linear search can make no progress along the search direction. If no progress can be made in the first conjugate gradient direction – that of steepest descent – then the algorithm is taken to have converged. A simple linear search algorithm is used : it first finds an "acceptable point" by stepsize reduction, using the same ideas as discussed in 2.6.2. Once an acceptable point has been found, there is enough information to fit a parabola to the restriction of the function on the search direction. The step length resulting from this inverse interpolation is used to generate a new trial point for the function. A starting step length k for the search is chosen to be $k = 1.0$. Factor 1.7 is used to multiply the best step length found in the linear search to increase the step. Recurrence formula (63) of Fletcher and Reeves [Fletcher-Reeves 1964] is used.

Given a function $f : R^n \longrightarrow R$ with its first partial derivatives and given an n-dimensional starting vector $\mathbf{xvec}$, routine *Cgmin* then finds the minimum. Parameter $intol$ is used as an initial value in computing the gradient test tolerance. Routine returns the location of the minimum $\mathbf{x}$ with the minimum value of the function $fmin$.

2.8 Descent to a minimum : Modified steepest descent method in multidimensions

Section 2.5 already provided an overview of *steepest descent method* – a simple minimizing algorithm by Cauchy. Its weakness is primarily that it generates linearly dependent search directions and thus results in very slow convergence. Conjugate gradients and variable metric methods were developed to correct this problem by transforming the gradient $\mathbf{g}$ of the function so that the search directions generated are linearly independent or, equivalently, are conjugate to each other to some positive definite matrix. This section describes a method which is primarily based on Cauchy's method and does not aim to produce linearly independent nor conjugate set of search directions. The strategy of choosing the search directions in *modified steepest descent method* is based on a heuristic idea and is closely related to so-called PARTAN-methods [Pierre 1969].

Consider the basic steepest descent algorithm :
STEEPEST DESCENT: Start at point $\mathbf{p}_0 \in R^n$. *As many times as needed, move from point* $\mathbf{p}_i$ *to the point* $\mathbf{p}_{i+1}$ *by minimizing along the line from* $\mathbf{p}_i$ *in the direction of the local downhill gradient* $-\mathbf{g}(\mathbf{p}_i)$.

For example, when finding minimum of a function of two dimensions in a long, narrow valley, each step of this algorithm starts off in the local gradient direction, perpendicular to the contour lines, and traverses a straight line until a local minimum of the function on that line is reached. At the point thus found, the contour line of the function through this point is tangent to the search direction. In order to improve the performance of this method, we should get rid of the "criss-crossing" or make it smoother. Our strategy is now as follows :
A MODIFIED STEEPEST DESCENT STEP : at each stage, keep track of points $\mathbf{p}_i$ *and* $\mathbf{p}_{i-1}, i \geq 1$ *and compare the results of two line minimizations : (i) line minimization made from* $\mathbf{p}_i$ *along the downhill gradient* $-\mathbf{g}(\mathbf{p}_i)$ *and call the new minimum point* $\mathbf{r}$ *(ii) line minimization made from* $\mathbf{p}_{i-1}$ *along the direction* $-\mathbf{p}_{i-1} + \frac{\mathbf{p}_i + \mathbf{r}}{2}$ *and call the new minimum point* $\mathbf{rr}$. *If* $f(\mathbf{rr}) < f(\mathbf{r})$ *then set* $\mathbf{p}_{i+1} \longleftarrow \mathbf{rr}$ *else set* $\mathbf{p}_{i+1} \longleftarrow \mathbf{r}$.

This strategy has proven to be much more efficient than the traditional steepest descent method, converging to the minimum almost as rapidly as the variable metric and conjugate gradients methods on average test problems. One can verify this by examining the test results in the last chapter of this study. The basic steps of this modified algorithm, compared with the steps taken by traditional steepest descent method, are shown in Figure 1. Figure 2 shows the routes of these two algorithms going down the valley of Rosenbrock's banana-shaped function $f(x, y) = 100(y - x^2)^2 + (x - 1)^2$ starting from point $(x, y) = (-1.2, 1.0)$. A three-dimensional plot of Rosenbrock's function is shown in Figure 19 in Appendix L.

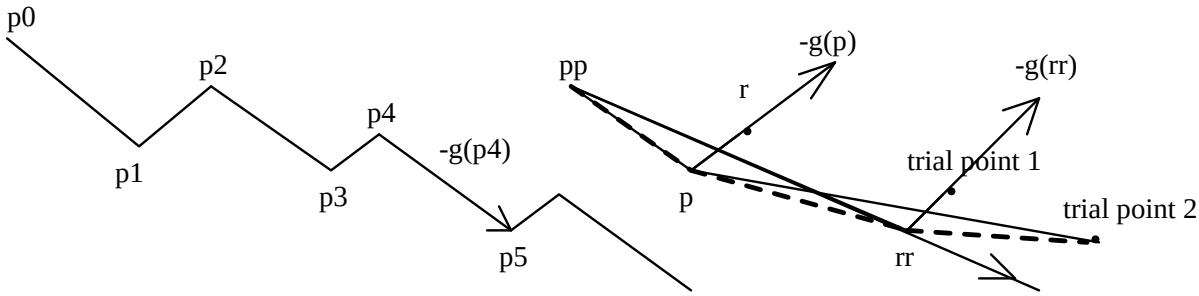

Figure 1: On the left is a series of typical steps $\mathbf{p}_0, \mathbf{p}_1, \ldots$, taken by the traditional steepest descent method. On the right is two subsequent steps by the modified algorithm : point $\mathbf{pp}$ is a result of the previous iteration (that is, $\mathbf{p}_{i-1}$ in the algorithm above) and $\mathbf{p}$ is the current point (that is, $\mathbf{p}_i$). Now suppose, that $f(\mathbf{rr}) < f(\mathbf{r})$ and thus the new point $\mathbf{p}_{i+1}$ will be $\mathbf{rr}$. As a result of line minimizations of the next step we would set $\mathbf{p}_{i+2} \longleftarrow \mathbf{trialpoint2}$. The minimization route is shown as a dashed line.

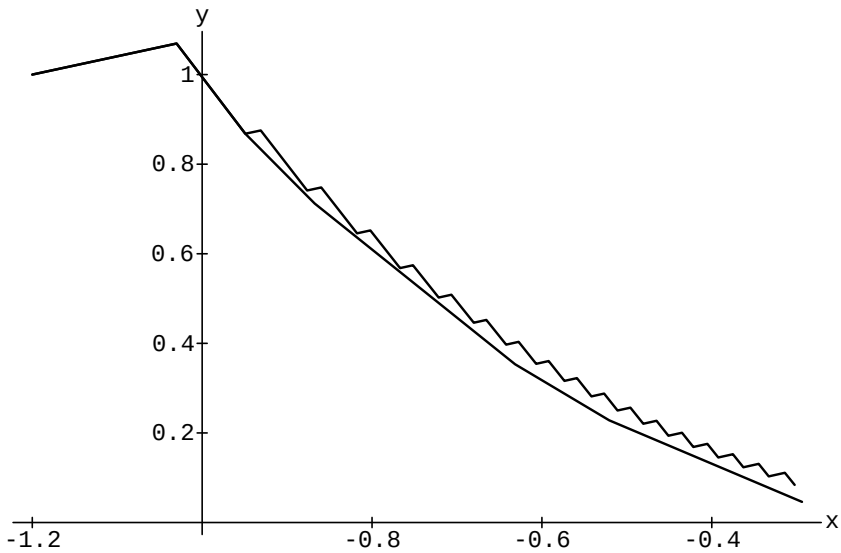

Figure 2: The routes of the traditional and the modified steepest descent method in the Rosenbrock banana-shaped valley. The modified method (lower route) takes only 7 steps while the traditional method takes 37 steps to get to the bottom of the valley.

2.8.1 Implemented routines : MSTEEPDESC

Routine *Msteepdesc* (in Appendix J) implements the minimizing of given n-dimensional function using modified steepest descent method :
Given a function $f : R^n \longrightarrow R$ with its first partial derivatives and given an n-dimensional starting point $\mathbf{p}$, routine *Msteepdesc* then finds the minimum. The first step is taken by minimizing along the line from $\mathbf{p}_0$ (given initially as a parameter $\mathbf{p}$) in the direction of the local downhill gradient $-\mathbf{g}(\mathbf{p}_0)$ thus achieving point $\mathbf{p}_1$. During the next iterations the strategy described above is used. When the routine has failed in decreasing the function

value by more than a pre-defined tolerance-value (value 0.0001 is used) in one iteration, or when the gradient is tolerably small (tolerance value 0.0005 is used), the routine is taken to have converged. Routine returns the location of the minimum $\mathbf{p}$ with the minimum value of the function $fret$.

3 Design and implementation of the minimization software package

This chapter introduces the design and implementation of the minimization software, including the user-interface for the minimization routines, as it was defined in section 1.2, and the minimization algorithms themselves (as they were defined in chapter 2). Installation and usage instructions are provided in the end of this chapter – in section 3.3.

3.1 Overall design of the minimization software

Section 1.2 already discussed how the minimization software should work. The minimization software should include (i) the actual minimization routines, (ii) an user-interface for entering the minimization problems, and (iii) a program which can generate 2- and 3-dimensional plots of functions.

As it was described in the previous chapter, the actual minimization algorithms were mostly adapted from literature or from supplementary distribution disks by Nash [Nash 1990] and Press et al [Press 1986]. The driver programs for these routines were also available from the same sources. All of these routines were then implemented by using C programming language. They all have the same kind of set of input and output parameters,as follows : given (i) pointers to the code containing the function and its gradient and given (ii) the starting value, which is almost always an n-vector, these routines search for a minimum, and if it is found, the minimum point and the function's value at the minimum is returned by a pointer.

Because the basic features of the user-interface were quite simple, it seemed straightforward to create a program which just prompts the user to enter the dimension and the definition of the function with the number of the desired method. Then, depending on the method, the program would ask the method-specific starting values. After this initialization the collected data could be given as input to the actual minimization routine which would pass, after having found the minimum, the results to the program doing the plotting of the function and showing the minimum point. With all the programming made in C, the whole task to create an interface between the user and the minimization routines would have been simple. This is because of the fact that the output from the interface would have been compatible with the input parameters of the minimization routines, assuming that the definition of the problem function is entered by using the syntax of C language. There was, however, the third component of the system – the plotting program – which caused problems of incompatibility.

The third component of the minimization software, the tool for plotting and also for minimization, was chosen to be *Mathematica* because of its strengths in visual presentation and symbolic calculation with wide collection of higher mathematical functions. *Mathematica* includes a high-level programming language with a syntax very similar to C programming language. For example, in *Mathematica* we would define a function $f(x, y) = \sqrt{x^2 + y^2}$ as follows :

```
f[x_,y_] := Sqrt[x^2 + y^2];
```

and inside of some routine written in C its appearance would be :

```
f = sqrt(pow(x,2) + pow(y,2));
```

A complete description of *Mathematica*'s syntax, naming conventions and mathematical functions can be found in *Mathematica : a system for doing mathematics by computer* by S. Wolfram [Wolfram 1991].

The main problem in combining the C programs and *Mathematica* was the presentation of the problem function definition :

- if the definition of the problem function would be entered in user-interface program by using syntax of C, it could not be used by plotting-routines of *Mathematica*, because there is no facility for "C -> Mathematica" syntax conversion, and alternatively,
- if the user would enter the definition of the function by using *Mathematica*'s syntax, thus enabling the plotting, could the *Mathematica*'s built-in routine CForm (which is for "Mathematica -> C" syntax conversion) be used to generate callable and compatible C code for the minimization routines ?

Because the first approach provided no solution by definition, the latter had to be examined. *Mathematica*'s conversion routine CForm was tested if it could properly translate all its equivalent standard C mathematical functions, that is, C language mathematical functions which appear also in *Mathematica*. As a result of a simple test *Mathematica* proved to convert (nearly) all of these functions succesfully, so the compatibility with the minimization routines was guaranteed after minor adjustments. The strategy *how* this conversion should be done, so that the minimization routines can use the function definition, is described in section 3.2.3. The implementation of the standard mathematical functions (with some additional extensions) can be found in Appendix K and the complete listing of functions known to user-interface is given in the end of this chapter – in section 3.3.5.

Figure 3 shows the general structure of the minimization software package. Section 3.2 discusses briefly the design of the individual software modules.

3.2 Design of the individual minimization software modules

This chapter discusses briefly the design of the individual software modules which, as a whole, implement the minimization software package.

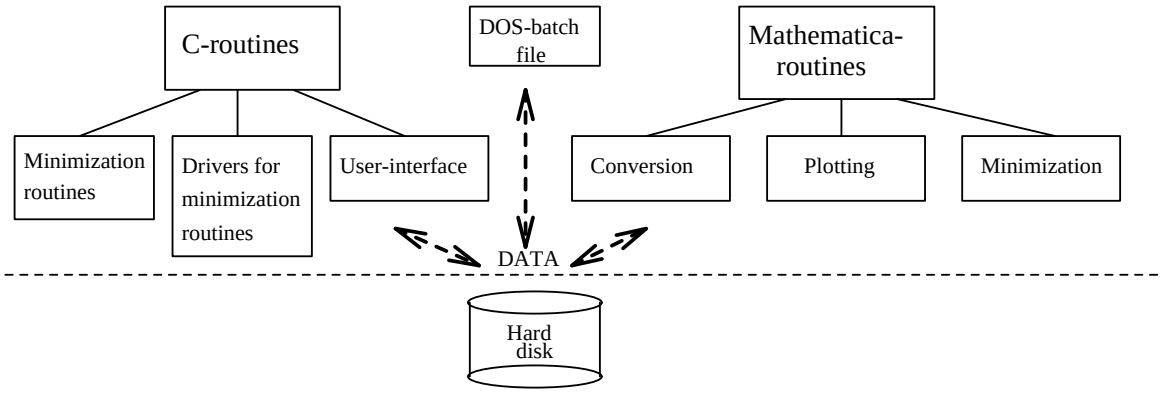

Figure 3: The software package consists of three separate parts : C language routines,Mathematica-routines and a DOS batch-file. The user-interface, and the actual minimization routines with their driver programs are implemented in C language. The conversion facility is programmed using Mathematica's programming language as are the plotting and minimization facilities. The DOS-batch file is a driver program for the whole system. All the communication between separate programs is done via hard disk.

3.2.1 The driver program for the software

The DOS-batch file (STARTME.BAT) mentioned above, handles the overall execution of the minimization software. It is just a programmed loop for calling the routines described in Figure 3 and it works basically as follows : call the user-interface program to start collecting the initial problem data, then call *Mathematica* to start the conversion of the function definition the user just entered, then call *Turbo C++* compiler and linker to combine the function (and possibly its gradient) definition with the desired minimization routine, and finally start the actual minimization. If the minimum was found, then call *Mathematica* to show the minimum and the traversed route. If the minimization routine did not find the minimum, then call *Mathematica* to try minimizing with its own minimization routine.

3.2.2 Modules of C language

The implementation of the actual minimization routines has already been described in chapter 2 and thus their requirements for input data and the information returned in output data is known. These routines cannot, however, work without a driver program. So, for each minimization routine there exists a driver program which, when executed, at first collects the data received from the user-interface and then calls the minimization routine by using the collected data as parameters. When the minimization has been done, the driver saves the results for later processing, that is plotting the function with its minimum and the route traversed. Modules of their name ending with DRV, e.g. BRENTDRV.C for routine Brent, are driver programs. Some of these driver programs set the initial values for precision required for minimization routines :

- The driver for routine *Brent* sets the fractional convergence tolerance $TOL = 0.0001$
- The driver for routine *Amoeba* sets the fractional convergence tolerance $FTOL = 0.0001$
- The driver for routine *Powell* sets the fractional convergence tolerance $FTOL = 0.0001$
- The driver for routine *Cgmin* sets the value used in computing the gradient test tolerance $INTOL = 0.0001$

The other minimization routines set the level of accuracy in their own code, as discussed in implementation sections of chapter 2.

The modules implementing the user-interface are MINCALC.C,METHODS.C and MCUTIL.C. Module MINCALC.C contains the main program, while METHODS.C has routines for collecting the method-specific data and MCUTIL.C contains the basic utility routines for processing user input and doing graphics. The installation diskette contains an executable file MINCALC.EXE, which is generated from these source modules.

Module MATHCORR.C (listed in Appendix K) implements the set of available mathematical functions, called by the *Mathematica* - generated module containing the converted problem function definition. Because *Mathematica* generates different kind of names for standard functions than *Turbo C++*, this module is used to interpret them correctly during the compilation. In addition to standard C mathematical functions, some implementations of higher transcendental functions are included here. The set of available functions can easily be extended by placing the new definitions in this file and their declarations in file MATHCORR.H. The names for the new routines should be exactly same as the corresponding routines in *Mathematica*.

Modules NRUTIL.C,PRMIN.C and VISUAL.C contain miscellaneous utility routines. The installation diskette contains executable files PRMIN.EXE and VISUAL.EXE, which are generated from the corresponding source modules.

3.2.3 Modules of Mathematica's programming language

The conversion "Mathematica -> C" of the user-supplied function definition is started by the DOS-batch program, as described above. It calls *Mathematica* which, by default, reads an initialization file INIT.M containing the conversion routine which has been initially copied from module CONVFUNC.M. This routine reads the data which the user-interface has collected from the user, including the definition of the problem function in *Mathematica*'s syntax. Then it calls *Mathematica*'s built-in CForm, which does the actual conversion. Because some minimization routines require gradient evaluation, the built-in D, which does partial differentation, is called in those cases to generate the definition for the gradient. Before the routine terminates, it generates a file FUNC.C which contains the complete definition of the function (and possibly its gradient) as C language code. For example, if the user enters a function of two variables :

```
f[x_,y_] = (x-2)^2 + (y-1)^2
```

the generated code in FUNC.C will be as follows :

```
#include <math.h>
#include "mathcorr.h"

float func(t)    /* The code of function func ... */
float t[];
{
 float x=t[1],y=t[2];
 return (float)(Power(-2 + x,2) + Power(-1 + y,2));
}

void dfunc(t,g)  /* ... and its gradient dfunc */
float t[],g[];
{
 float x=t[1],y=t[2];
 g[1] = (float)2*(-2 + x);
 g[2] = (float)2*(-1 + y);
}
```

Now the file FUNC.C can be compiled and linked with the actual minimization routine and its driver program, thus generating an executable file finally doing the minimization.

The following *Mathematica* -language modules contain plotting routines : DOPICT.M and PLOTFUNC.M for generating plot of the function, ROUPLOT.M for plotting the route traversed by the minimization algorithm, PLOTINIT.M for user-defined settings affecting the plotting (described in the next section) and VISUAL.M for so-called Visual Bracketing with Brent's method in one-dimension. The simple strategy for visually bracketing the minimum goes as follows : when the user is asked to define the interval (ax, bx) (in user-interface program) used as an initial guess for bracketing algorithm, the program then asks if these values should be *at first* used as limits for generating a plot of the problem function ; if the user wishes to do that, the plot of a function is shown, and thus the user has a possibility to redefine the original values, which are then used as parameters for actual bracketing algorithm.

Module TRYMATH.M contains the logic for calling *Mathematica*'s own minimization routine and plotting routines.

3.3 User's guide

This section provides information about installing and using the minimization software package.

3.3.1 Software and harware requirements

The minimization software package is stored on a 3.5" disk of 1.44 megabytes capacity. It runs on the 386-based IBM PC family of computers (including all true IBM compatibles)

with a VGA monitor, a hard disk drive and one 1.44 MB drive. The software package requires operating system MS-DOS 3.3 or higher, *Mathematica* version 1.2 (or higher) and Borland *Turbo C++* version 1.01 (or higher). It is recommended, that the lowest possible versions of *Mathematica* and *Turbo C++* are used, although the software may run with higher versions, but may sometimes cause unexpected results. It is also recommended, that they are both installed under root directory of hard disk partition C in your system. The amount of required RAM depends on which version of *Mathematica* is used, usually at least 1024 KB extended memory should be available. Minimization algorithms use floating-point operations extensively, so the use of an 80x87 math coprocessor is recommended. This software has no mouse support.

3.3.2 Installation of the software package

This software has no automatic installation program, so the user is completely responsible for doing the following installation steps correctly :

- Create a DOS-subdirectory MINCALC under the root of some available logical disk (disk C is recommended), and set it as the default directory, for example

```
C:\> MKDIR MINCALC
C:\> CD MINCALC
```

- Insert the installation diskette in drive A and copy all the files from it to the directory just created

```
C:\MINCALC> COPY A:\*.* *.*
```

- Copy the VGA-graphics device driver file EGAVGA.BGI from the *Turbo C++* BGI-directory (may be different in your system than the following directory)

```
C:\MINCALC> COPY C:\TC\BGI\EGAVGA.BGI *.*
```

If you have installed *Mathematica* in directory MATH in disk C, that is under

```
C:\MATH
```

in your system, and Turbo C++ under

```
C:\TC
```

you don't need to make the following modifications.

If either *Mathematica* or *Turbo C++*, (or both) are installed in different directories or disks than just mentioned, you have to check and possibly modify the directory-specifications appearing in the following files :

```
TURBOC.CFG
LISTLIBS.
LISTOBJS.*
PLOTINIT.M
RUNME.BAT
```

The first three files contain references to *Turbo C++* directories, the fourth file (PLOTINIT.M) contains references to *Mathematica*'s directories and the last one (RUNME.BAT) to both of these. Modifications to files refering to *Turbo C++* directories should be trivial, and these files contain thus no explanatory comments. On the contrary, files PLOTINIT.M and RUNME.BAT contain detailed comments and usage instructions.

The minimization routines are, by default, compiled and linked with the math coprocessor option turned on. If your system does not have a math coprocessor, remove the option *-f87* from file TURBOC.CFG and replace the reference to the floating-point library *fp87* by *emu* in file LISTLIBS..

3.3.3 Initialization and set-up

Once the steps of the previous section are completed, you may run the initialization batch file RUNME.BAT to set the DOS-environment variables. Be sure to enter the following command in MINCALC subdirectory :

```
C:\MINCALC> RUNME
```

This batch file should be run at least once per each DOS-session. Then run the batch file MAKEOBJS.BAT to compile the source code of the minimization routines into object files :

```
C:\MINCALC> MAKEOBJS
```

This batch file should be run only once, unless the source code is manually modified or any of the object files intentionally deleted.
The minimization package uses now the following default settings :

- The function definition entered via user-interface is converted by *Mathematica* at the beginning of processing of each minimization problem.
- The plot of the problem function with the traversed minimization route is generated after minimum is found.
- *Mathematica* is started to search the minimum of a function only when the actual minimization routine has failed.
- No statistical data will be collected.

These settings can be changed by modifying the DOS-environment variables in file RUNME.BAT, which contains detailed modification instructions.
The following default settings (effecting the plotting) can be changed by modifying the file PLOTINIT.M :

- The scale of the picture (described in detail in file PLOTINIT.M)
- If *Mathematica* is used for minimization, it does not plot the function.
- The minimization route traversed by the minimization routine is plotted (if it contains more than just one routepoint).
- The contour plot of function with two variables is not shown.

3.3.4 Usage of the interface

The purpose of the user-interface is to collect data of each minimization problem. It is started via DOS-batch file STARTME.BAT (described in section 3.2.1) by entering its name after the command prompt :

```
C:\MINCALC> STARTME
```

This command actually starts the whole minimization process which can be controlled via user-interface.

The main window of the user-interface now appears on the screen prompting you to enter the dimension of the problem. You can now get help by pressing <F1> or exit by pressing <ESC>, as in every stage of the user-interface program. Pressing the <ESC>-key, though, elsewhere than in the first prompt, just clears the contents of the current part of the screen rather than immediately exiting the program. If you press <F1> in any stage, a help text concerning the current prompt appears on the screen. To get rid of the help window, just press any key. After any prompt you may use the normal typing keys to enter the desired values. Note, that pressing the arrow keys has no effect – the only way to move horizontally is to use typing keys or the <BACKSPACE> key.

Now choose the dimension as integer value 1, 2 or 3 and press <RETURN>. Then you are prompted to enter the definition of the function to be minimized. Write the definition by using the syntax and names for functions exactly as they would appear in *Mathematica*, and press <RETURN>. Note, that you *must* use x,y and z as the variable names when entering the definition. For example, if you have chosen the dimension to be 2, then the problem function has to be defined by using *two* variables x and y. Suppose now, that the problem function is $f(x, y) = 100(y - x^2)^2 + (x - 1)^2$. After entering its definition the screen would appear as follows :

```
Dimension : 2
Definition of function to be minimized ; f[x,y] =
100 * (y - x^2)^2 + (x - 1)^2
Method to be used in minimization :
```

Note, that you can also use previously defined functions as follows : press <F1> when you are at the second prompt to see the previous definitions, then press any key to exit help, and then enter character @ followed by the number of the desired function shown in help window. You can also enter any file specification containing function definition after character @ (see section 3.3.6).

The next prompt is for choosing the method to be used in minimization. The possible alternatives are (as shown in help window) :

```
Methods for 1-dimensional functions
===================================
1. Parabolic interpolation (Brent)

Methods for N-dimensional functions
===================================
2. Downhill simplex (Nelder-Mead)
3. Direction set (Powell)
4. Variable metric (Fletcher-Nash)
5. Modified steepest descent
6. Conjugate gradients (Fletcher-Reeves)
```

Now choose any integer value from range `[1..6]` to select the desired minimization method and press <RETURN>. Note, that you can use method number one (Brent's method) only for one-dimensional functions.

The next stage in user-interface is to enter the method-specific starting values, that is, depending on your choice of method, you are prompted to give the initial starting values characteristic for each minimization method. The precision for these values should be limited to eight decimal digits.

- For Brent's method, you are at first asked if you wish to use visual bracketing, which is a simple method for visually determining the interval bracketing a minimum of one-dimensional function. If you answer "Y" for *Yes* you are then prompted to give two real values ax and bx which define the interval (ax, bx) to be used in plotting the problem function. After a while the plot of the function is shown : press <RETURN> after viewing the picture and enter the (possibly new) values for ax and bx when prompted. These will be used as initial values for the bracketing algorithm. If you answered "N" (or just pressed <RETURN>), you are then prompted to enter the values ax and bx without the possibility to view the plot at first. Note, that visual bracketing is not possible when you have defined the function as an external file.

- For downhill simplex method, enter real-valued coordinates for the initial simplex of $n + 1$ vertices, where n is the dimension of the problem function. Each vertice is an n-vector. For example, to define a starting simplex for 2-dimensional function, you might enter :

  ```
  VERTICE #1
  Coordinate #1 : 0.0
  Coordinate #2 : 0.0
  VERTICE #2
  Coordinate #1 : 1.0
  Coordinate #2 : 0.0
  VERTICE #3
  Coordinate #1 : 0.0
  Coordinate #2 : 1.0
  ```

- For Powell's method, enter first n real-valued coordinates for the initial starting point. Then you are asked, if the n unit vectors should be used as an initial set of directions : you may answer "N" for *No*, and "Y" or <RETURN> for *Yes*. If you don't wish to use them, you are then prompted to enter n vectors, each of them being an n-vector. For example, to define a starting point and the inital set of directions for 2-dimensional function, you might enter :

```
INITIAL STARTPOINT
Coordinate #1 : 1.0E-3
Coordinate #2 : 1.2
Do You wish to use unit-vectors as initial directions [Yes] ? : N
INITIAL DIRECTION #1
Coordinate #1 : 1.0
Coordinate #2 : 2.0
INITIAL DIRECTION #2
Coordinate #1 : 2.0
Coordinate #2 : 0.0
```

- For the rest of the methods, you are prompted for an initial startpoint, an n-vector, which defines the initial coordinates for the algorithm to start searching the minimum. For example, to define a starting point for 3-dimensional function to be minimized by using the *modified steepest descent method*, you might enter :

```
INITIAL STARTPOINT
Coordinate #1 : -2.3
Coordinate #2 : 4.6
Coordinate #3 : 11.4
```

When the user-interface program has collected all the initial problem data, the user is notified of the following events : starting of *Mathematica* to convert the definition of the function, compilation and linking of the minimization routines and the status (i.e. success or failure) of the minimization routine. Note, that if the computational time spent in minimization exceeds 90 seconds, the routine is taken to have failed and the user is notified also of this event.

In case of success, the user is informed at first how much computational time was spent. Then the minimum and the gradient at the minimum (when available) is displayed with a 2- or 3-dimensional plot of function, whenever plotting is possible. If the plotting is not possible, the user receives a message, which reports the results in textual form. The value of function at the minimum, the minimum point and the value of gradient at the minimum (when available), are shown in at most four-digit decimal precision as follows :

```
Minimum found = 0.5 at {1.0,2.0} ; Gradient = {0.0001,0.0001}
```

After the minimum is shown, the user can proceed by pressing <RETURN> to view the minimization route : when the route is plotted, a label appears above the picture informing how many route points (if any) the route consists of. The route will be not plotted, if it contains only one route point. The minimization route of an n-dimensional function consists of subsequent points, that is n-vectors, which the routine has traversed during its

search for a minimum. Routine *Amoeba* also records its route as subsequent n-vectors, consisting of mean values of the vertices, rather than saving all the vertices of each simplex. For functions of dimension $n = 2, 3$ the route can be then plotted as it is, that is, as a 2- or 3-dimensional plot. The minimization route of an one-dimensional function, which is just a series of x-coordinates, is always plotted by pairs of points $(x, f(x))$.

After the plot of function or traversed route is shown, the user can proceed by pressing <RETURN>.

In case of failure of the original minimization routine, the user is also notified why the routine failed (e.g. division by zero or time-out). If *Mathematica* is then used for minimization, a message appears on the screen informing the user of the initial starting values used for *Mathematica*'s minimization routine. If *Mathematica* succeeds in finding the minimum, then the minimum is shown either with or without a plot of the function, depending on the settings of file PLOTINIT.M. The minimum is shown in the same format as above (omitting the gradient, though).

When the software has finished with the processing of the current minimization problem (which is, by default, after route-plotting) the user is returned to the user-interface level, prompting for a new problem.

3.3.5 List of functions known to user-interface

The mathematical functions which are known to user-interface, that is, the functions which can be used when defining the problem function, are basically same than those implemented in C language and listed in Appendix K. These functions are (listed in alphabetical order) :

```
Abs[x]
ArcCos[x]
ArcSin[x]
ArcTan[x]
ArithmeticGeometricMean[a,b]
BesselJ[n,x]  /* n = -1,0,1 */
Ceiling[x]
Cos[x]
Cosh[x]
Cot[x]
EllipticE[r]
EllipticK[r]
Floor[x]
Hypergeometric2F1[a,b,c,r]
Log[x]
Mod[x,y]
Power[x,y]    /* usage of notation xˆy is recommended */
Sec[x]
Sin[x]
Sinh[x]
Sqrt[x]
Tan[x]
Tanh[x]
```

A complete description of these functions can be found in the appendix of *Mathematica* user's guide [Wolfram 1991]. Note, that the symbols for basic arithmetic operators $+, -, *, /$ and ^ should be used as defined in [Wolfram 1991].

3.3.6 Using externally defined functions

As it was discussed in 3.3.4, you can enter any file specification containing function definition after character @ when defining the problem function in the user-interface program. The external definition for an n-dimensional ($n = 1, 2, 3$) function to be minimized with previously described methods (except *Brent*'s method) should generally be of following format :

```
#include <math.h>

float func(t)
float t[];
{
 float value;

 value = <function evaluation>
 return(value);
}

void dfunc(t,g)
float t[],g[];
{
 for (i=1;i<=n;i++)
     g[i] = <gradient evaluation>
}
```

where <function evaluation> in routine $func$ contains references to t, which is an n-vector – the point where the function is to be evaluated. The routine $dfunc$ should do the <gradient evaluation> refering to values of vector t and return the gradient vector g, which is also an n-vector. Note, that routine $dfunc$ has no effect when minimizing with methods which do not require gradient evaluation (that is, *Brent*,*Amoeba* and *Powell*). Creating a definition for one-dimensional function to be minimized with *Brent*'s method should be trivial : the parameter t for routine $func$ is then of simple floating-point type. A sample file PARABOLF.C contains an external function definition to be used with multidimensional methods.

4 Testing of the minimization algorithms

This chapter introduces the techniques for testing the minimization algorithms, describes the set of test problems used, and discusses the test results.

4.1 Functions used in testing

The following set of test functions will be used in testing the minimization algorithms included in the minimization software package. Test functions are labeled in respect to their dimension : one-dimensional functions will be referred by numbers 10..14, two-dimensional functions by 20..25, and three-dimensional functions by 30..32.

$$
\begin{aligned}
10 \;\; f(x) &= x^3 - 2x + 5 \\
11 \;\; f(x) &= \sin\tan x \\
12 \;\; f(x) &= \frac{-\sin x}{x} \\
13 \;\; f(x) &= \frac{1}{x} + K(x) + K^2(x) \\
14 \;\; f(x) &= x^4 - 12x^3 + 47x^2 - 60x \\
20 \;\; f(x,y) &= (x-2)^4 + (x-2)^2y^2 + (y+1)^2 \\
21 \;\; f(x,y) &= 100(x^2 - y)^2 + (1-x)^2 \\
22 \;\; f(x,y) &= 100((100x)^2 - \frac{y}{100})^2 + (1-100x)^2 \\
23 \;\; f(x,y) &= y^3 - y(x - \frac{1}{\sqrt{3}})^2 + x^3 - x - y \\
24 \;\; f(x,y) &= \tan^2 x + \sin^2 \frac{x}{y} \\
25 \;\; f(r,\varphi) &= r^2 + r\sin^2 \frac{\varphi}{r} \\
30 \;\; f(x,y,z) &= 100(x^2 - y)^2 + (1-x)^2 + 100(1-z)^2 \\
31 \;\; f(x,y,z) &= 3 + (x-1)^2 + (y-2)^2 + (z+5)^2 \\
32 \;\; f(x,y,z) &= 1 - J_0(x - \frac{1}{2})J_0(y - \frac{1}{2})J_0(z - \frac{1}{2})
\end{aligned}
$$

where K is the complete elliptic integral of the first kind and J_0 is the Bessel function of the first kind. Functions number 21 and 22 are Rosenbrock's banana-shaped functions of different scales in two-dimension and number 30 is an extended Rosenbrock function in three-dimension. Function number 25 defines a parabolic surface in parametric form, and thus cannot be entered directly via user-interface of the minimization software like the other test functions, as follows :

```
10 f[x] = x^3 - 2x + 5
11 f[x] = Sin[Tan[x]]
12 f[x] = -Sin[x]/x
13 f[x] = 1/x + EllipticK[x] + (EllipticK[x])^2
14 f[x] = x^4 - 12x^3 + 47x^2 - 60x
20 f[x,y] = (x-2)^4 + y^2(x-2)^2 + (y+1)^2
21 f[x,y] = 100(x^2 - y)^2 + (1-x)^2
22 f[x,y] = 100((100x)^2 - y/100)^2 + (1-100x)^2
23 f[x,y] = y^3 - y(x - 1/Sqrt[3])^2 + x^3 - x - y
24 f[x,y] = (Tan[x])^2 + (Sin[x/y])^2
25 f[x,y] = @parabolf.c
```

```
30 f[x,y,z] = 100(x^2 - y)^2 + (1-x)^2 + 100(1-z)^2
31 f[x,y,z] = 3 + (x-1)^2 + (y-2)^2 + (z+5)^2
32 f[x,y,z] = 1 - BesselJ[0,x-0.5]*BesselJ[0,y-0.5]*BesselJ[0,z-0.5]
```

These functions have the following (at least local) minima :

$$
\begin{aligned}
10\ \ x_* &= \sqrt{\frac{2}{3}}\ ,\ \ f(x_*) \simeq 3.9113 \\
11\ \ x_* &= \arctan(\frac{-\pi}{2} + 2n_1\pi) + n_2\pi\ \ ,\ \ n_1, n_2 \in Z\ \ ,\ \ f(x_*) = -1 \\
12\ \ x_* &= 0\ \ ,\ \ f(x_*) = -1 \\
13\ \ x_* &\simeq 0.5005\ \ ,\ \ f(x_*) \simeq 7.2917 \\
14\ \ x_* &\simeq 0.9435\ \ ,\ \ f(x_*) \simeq -24.0573\ \ \textit{or} \\
x_* &\simeq 4.6010\ \ ,\ \ f(x_*) \simeq -1.7664 \\
20\ \ \mathbf{x}_* &= (2,-1)\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
21\ \ \mathbf{x}_* &= (1,1)\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
22\ \ \mathbf{x}_* &= (\frac{1}{100}, 100)\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
23\ \ \mathbf{x}_* &= (\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})\ \ ,\ \ f(\mathbf{x}_*) = \frac{-4}{3\sqrt{3}} \\
24\ \ \mathbf{x}_* &= (n\pi, n)\ \ ,\ \ 0 \neq n \in Z\ \ \textit{or} \\
\mathbf{x}_* &= (0,t)\ \ ,\ \ 0 \neq t \in R\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
25\ \ \mathbf{x}_* &= (0,0)\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
30\ \ \mathbf{x}_* &= (1,1,1)\ \ ,\ \ f(\mathbf{x}_*) = 0 \\
31\ \ \mathbf{x}_* &= (1,2,-5)\ \ ,\ \ f(\mathbf{x}_*) = 3 \\
32\ \ \mathbf{x}_* &= (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})\ \ ,\ \ f(\mathbf{x}_*) = 0
\end{aligned}
$$

where the minimum of function number 12 should be, in fact, expressed in terms of limiting value of f when x approaches x_*. Despite of the fact that this function cannot be evaluated at its precise "minimum", the algorithms are expected to isolate it without problems.

4.2 How the tests are carried out : minimization methods vs. each other

The previously described minimization algorithms can be grouped into several categories by common features : algorithms which are solely for one-dimensional minimization, algorithms which can solve multidimensional problems, algorithms which make use of the gradient of the function or those which don't do that. There are thus many possibilities to group and test the methods. Normally we would group them at first by some common

feature, then test the methods of the same group with the same set of test problems, and finally compare the results. How the results are then compared – is often a difficult question. Which method would be the "best" : a method which used a lot of computational time but found the minimum with the best possible accuracy, or a method which found the minimum with a minimal computational cost but indicated poor accuracy ?

The purpose of the testing described in the following sections is *not* to find the best possible multi-purpose minimization algorithm, but to generally observe the performance of each algorithm on the various test problems. The measure of efficiency is the amount of CPU-time spent by each algorithm and ability to find reasonably accurate (at least local) minimum. Minimization algorithms are grouped into three categories by their capability to minimize one-dimensional functions, multidimensional functions, or both, and tested as follows :

- Testing the one-dimensional method (i.e. *Brent*'s method) with one-dimensional functions
- Testing the multidimensional methods with one-dimensional functions
- Testing the multidimensional methods with multidimensional functions

The whole testing can be now carried out by using the minimization software normally : entering the test functions and starting values via user-interface and collecting the results with the built-in facility (enabled from set-up file RUNME.BAT). The following sections describe how the actual test problems are defined for each test group.

4.2.1 Testing the one-dimensional method

This section describes the testing of the *Brent*'s method with one- dimensional functions (numbers 10..14 as described above).

Because of the two different bracketing algorithms *Mnbrak1* and *Mnbrakn* can now be used as a subroutine for routine *Brent*, it is obvious, that their performance should be measured. This approach divides the one-dimensional test-case in two parts : (i) testing routine *Brent* with *Mnbrak1* and (ii) testing routine *Brent* with *Mnbrakn*. Both of these are tested with the following initial intervals (ax, bx), $ax, bx \in R$ for each test function : (note, that the initial values for test functions are labeled with lowercase letters from now on)

```
10.a ax =   -9.0  bx =  -7.0
10.b ax =   -1.0  bx =   2.0
10.c ax =   15.0  bx =  25.0

11.a ax =  -10.0  bx =  -1.0
11.b ax =    0.0  bx =   0.5
11.c ax =    1.0  bx =   2.0

12.a ax = -100.0  bx = -50.0
12.b ax =   -0.1  bx =   0.1
12.c ax =   10.0  bx =  10.5
```

```
12.d ax =   -0.1  bx =   0.2
12.e ax =   -1.0  bx =   5.0

13.a ax =   0.05  bx =   0.1
13.b ax =   0.01  bx =  0.99
13.c ax =   0.45  bx =  0.55

14.a ax =  -10.0  bx =  10.0
14.b ax =    3.0  bx =   6.0
14.c ax =    4.0  bx =   6.0
```

where the initial intervals for functions 10,11 and 12 are chosen to be critical, that is, to lead the bracketing algorithms either into infinity or near singularities.

4.2.2 Testing the multidimensional methods with one-dimensional functions

This section describes the testing of the *downhill simplex*, *direction set* and *variable metric* methods with one-dimensional functions (numbers 10..14 as described above). The actual routines to be tested are *Amoeba*,*Powell* and *Vmmin*, respectively. The rest of the methods which make use of gradient are omitted because of the fact that *Mathematica* cannot generate definition for gradient of function number 13, and thus this test-case would be anyhow incomplete.

The downhill simplex method is tested by using the following initial simplexes with vertices $p_1 \in R$ and $p_2 \in R$:

```
10.a p1 =   -9.0  p2 =  -7.0
10.b p1 =   -1.0  p2 =   2.0
10.c p1 =   15.0  p2 =  25.0
10.d p1 =    1.0  p2 =   3.0

11.a p1 =  -10.0  p2 =  -1.0
11.b p1 =    0.0  p2 =   0.5
11.c p1 =    1.0  p2 =   2.0
11.d p1 =   -5.0  p2 =   2.0

12.a p1 = -100.0  p2 = -50.0
12.b p1 =   -0.1  p2 =   0.1
12.c p1 =   10.0  p2 =  10.5
12.d p1 =   -0.1  p2 =   0.2
12.e p1 =   -1.0  p2 =   5.0

13.a p1 =   0.05  p2 =   0.1
13.b p1 =   0.01  p2 =  0.99
13.c p1 =   0.45  p2 =  0.55

14.a p1 =  -10.0  p2 =  10.0
14.b p1 =    3.0  p2 =   6.0
14.c p1 =    4.0  p2 =   6.0
```

where the vertices of the initial simplexes are adapted from the initial values for Brent's method.

The direction set (Powell's) method and the variable metric method are tested by using the following initial starting points $p \in R$:

```
10.a p =  -8.0
10.b p =   0.5
10.c p =  20.0

11.a p =  -5.5
11.b p =  0.25
11.c p =   1.5

12.a p = -75.0
12.b p =   0.0
12.c p = 10.25
12.d p =  0.05
12.e p =   2.0

13.a p = 0.075
13.b p =   0.5
13.c p =  0.75

14.a p =   0.0
14.b p =   4.5
14.c p =   5.0
```

which are mean values of initial points ax,bx for Brent's method. Unit vectors are used as an initial set of directions for Powell's method.

4.2.3 Testing the multidimensional methods with multidimensional functions

This section describes the testing of the *downhill simplex*, *direction set*, *variable metric*, *conjugate gradients* and *modified steepest descent* methods with multidimensional functions (numbers 20..32 as described above). The actual routines to be tested are *Amoeba*,*Powell*, *Vmmin*, *Cgmin* and *Msteepdesc*, respectively.

The downhill simplex method is tested by using the following initial simplexes with vertices $\mathbf{p}_i \in R^2 \;\; i = 1, \ldots, 3$ and $\mathbf{p}_i \in R^3 \;\; i = 1, \ldots, 4$:

```
20.a p1 = (0.0,0.0) p2 = (1.0,0.0)  p3 = (0.0,1.0)

21.a p1 = (0.0,0.0) p2 = (1.0,0.0)  p3 = (0.0,1.0)
21.b p1 = (0.0,0.0) p2 = (-1.2,0.0)  p3 = (0.0,1.0)
21.c p1 = (0.0,0.0) p2 = (0.5,0.0)  p3 = (0.0,0.5)

22.a p1 = (0.0,0.0) p2 = (1.0,0.0)  p3 = (0.0,1.0)

23.a p1 = (0.0,0.0) p2 = (1.0,0.0)  p3 = (0.0,1.0)

24.a p1 = (1.0,1.0) p2 = (2.0,1.0)  p3 = (1.0,2.0)

25.a p1 = (0.0,0.0) p2 = (1.0,0.0)  p3 = (0.0,1.0)
25.b p1 = (1.0,0.0) p2 = (1.0,2.0)  p3 = (2.0,0.0)
```

```
30.a p1 = (0,0,0) p2 = (1,0,0)     p3 = (0,1,0)  p4 = (0,0,1)
30.b p1 = (0,0,0) p2 = (-1.2,0,0)  p3 = (0,1,0)  p4 = (0,0,-1.2)

31.a p1 = (0,0,0) p2 = (1,0,0)  p3 = (0,1,0)  p4 = (0,0,1)

32.a p1 = (0,0,0) p2 = (1,0,0)  p3 = (0,1,0)  p4 = (0,0,1)
32.b p1 = (1,1,1) p2 = (2,1,1)  p3 = (1,2,1)  p4 = (1,1,2)
```

where the initial simplexes b for functions 21 and 30 (Rosenbrock banana-shaped function) imitate the normally used standard starting points $\mathbf{p} = (-1.2, 1.0)$ and $\mathbf{p} = (-1.2, 1.0, -1.2)$, which force algorithms to traverse the "valley" both in depth and length.

The rest of the multidimensional methods are tested by using the following initial starting points $\mathbf{p} \in R^n$, $n = 2, 3$:

```
20.a p = (1.0,2.0)
20.b p = (0.0,0.0)
20.c p = (-10.0,15.0)

21.a p = (-1.2,1.0)
21.b p = (0.5,0.5)
21.c p = (6.39,-0.221)

22.a p = (-1.2,1.0)
22.b p = (0.5,0.5)
22.c p = (6.39,-0.221)

23.a p = (3.0,0.0)
23.b p = (-1.0,0.0)
23.c p = (2.0,3.0)

24.a p = (1.0,1.0)
24.b p = (-1.0,3.0)
24.c p = (15.0,-10.0)
24.d p = (1.0,0.0001)

25.a p = (-1.0,1.0)
25.b p = (-4.0,5.0)
25.c p = (10.0,25.0)

30.a p = (-1.2,1.0,-1.2)
30.b p = (0.5,0.5,0.5)
30.c p = (6.39,-0.221,6.39)

31.a p = (1.0,1.0,1.0)
31.b p = (-1.0,-2.0,5.0)
31.c p = (8.0,-12.0,0.0)

32.a p = (0.0,0.0,0.0)
32.b p = (3.0,2.0,1.0)
32.c p = (20.0,-18.0,4.0)
```

where unit vectors are used as an initial set of directions for Powell's method. Two additional standard starting points $\mathbf{p} = (6.39, -0.221)$ and $\mathbf{p} = (6.39, -0.221, 6.39)$ are used here in minimizing the variations of Rosenbrock's function (numbers 21,22 and 30). The y-coordinates of the initial points a and b for function number 23 are set to zero to (possibly) initialize a problem situation when the algorithms try to minimize along the line $y = 0$. Because function number 25 has infinite amount of local minimizers along any line going through the global minimum, the initial starting points are chosen in such a way that the line minimization routines are expected to fail (or at least indicate poor accuracy).

4.3 Test results

All the test-cases described in the previous sections were run under MS-DOS 5.00 operating system on a 80386SX-based (16 MHz) personal computer without math coprocessor. The results they produced (such as the found minimum and the route traversed by the algorithm) were collected into data files, which are also available in the installation diskette, with extension *.STA. The results of each test-case are described in the next three sections.

4.3.1 Results of testing the one-dimensional method

Testing of routine *Brent* with bracketing done by *Mnbrak1* produced the following results on one-dimensional functions 10..14 (with the initial intervals defined in section 4.2.1) :

Function 10 The local minimum $x_* = \sqrt{\frac{2}{3}}$ was found with all the initial intervals a, b, c with precision ± 0.0001. The computational time spent in bracketing was on average 63 % of the total elapsed CPU-time (59.946 seconds).

Function 11 Three distinct global minima (where $f(x_*) = -1$) were found with the initial intervals a, b, c with precision ± 0.0001. The computational time spent in bracketing was on average 85 % of the total elapsed CPU-time (4.725 seconds).

Function 12 The global minimum $x_* = 0$ was found with initial intervals d and e with precision ± 0.0001. Two distinct local minimizers were found with the initial intervals a and c. The bracketing algorithm failed with the initial interval b (division by zero). The computational time spent in bracketing was on average 40 % of the total elapsed CPU-time (3.407 seconds).

Function 13 The global minimum $x_* \simeq 0.5005$ was found with all the initial intervals a, b, c with precision ± 0.0001. The computational time spent in bracketing was on average 63 % of the total elapsed CPU-time (57.582 seconds).

Function 14 The global minimum $x_* \simeq 0.9435$ was found with initial intervals a and c and the local minimum $x_* \simeq 4.6010$ with initial interval b ; all with precision ± 0.0001. The computational time spent in bracketing was on average 32 % of the total elapsed CPU-time (0.925 seconds). The minimization route after bracketing

with initial interval a is shown in Figure 4. Figure 5 shows the graph of the function together with the route.

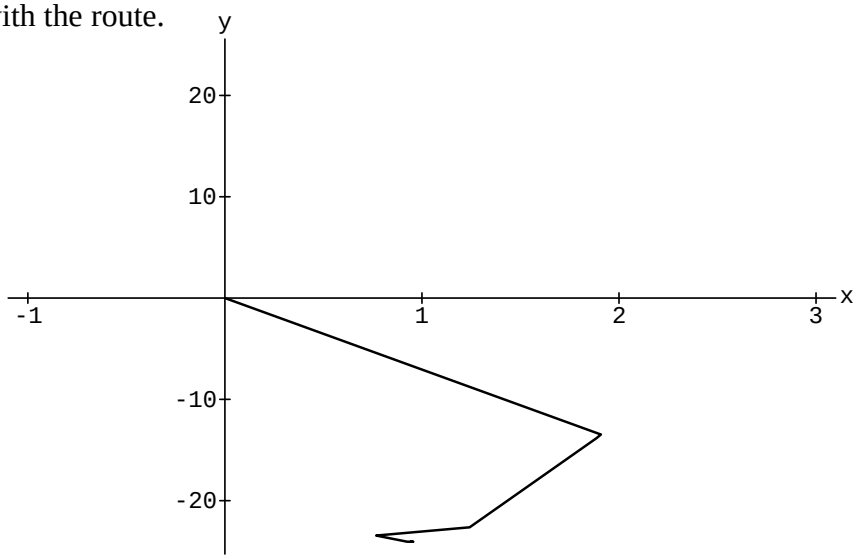

Figure 4: The minimization route of routine *Brent* with problem 14.a consists of 10 routepoints. Initially the minimum $x_* \simeq 0.9435$ is bracketed by triplet $a < b < c$ where $a = -10$,$b = 0$ and $c = 10$.

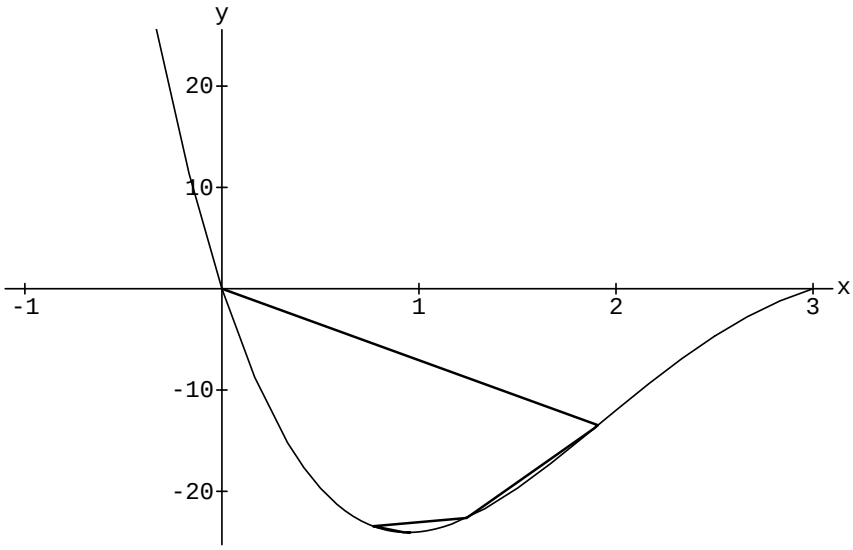

Figure 5: The minimization route of routine *Brent* (problem 14.a) with the graph of function $f(x) = x^4 - 12x^3 + 47x^2 - 60x$

Testing of routine *Brent* with bracketing done by *Mnbrakn* produced the following results on one-dimensional functions 10..14 (with the initial intervals defined in section 4.2.1) :

Function 10 The bracketing algorithm failed with all the initial intervals a, b, c (floating-point overflow).

Function 11 Two distinct global minima (where $f(x_*) = -1$) were found with the initial intervals a, b, c with precision ± 0.0001. The computational time spent in bracketing was on average 9 % of the total elapsed CPU-time (0.605 seconds).

Function 12 The global minimum $x_* = 0$ was found with initial values b, d, e with precision ± 0.0001. Two distinct local minimizers were found with the initial intervals a and c. The computational time spent in bracketing was on average 10 % of the total elapsed CPU-time (1.375 seconds).

Function 13 The global minimum $x_* \simeq 0.5005$ was found with all the initial intervals a, b, c with precision ± 0.0001. The computational time spent in bracketing was on average 22 % of the total elapsed CPU-time (1.703 seconds).

Function 14 The global minimum $x_* \simeq 0.9435$ was found with all the initial intervals a, b, c with precision ± 0.0003. The computational time spent in bracketing was on average 22 % of the total elapsed CPU-time (0.770 seconds).

The complete results of this test-case can be found in file BRENT.STA in the installation diskette.

4.3.2 Results of testing the multidimensional methods with one-dimensional functions

Testing of routine *Amoeba* produced the following results on one-dimensional functions 10..14 (with the initial simplexes defined in section 4.2.2) :

Function 10 Routine failed with the initial simplexes a, b, c (floating-point overflow), but found the local minimum $x_* = \sqrt{\frac{2}{3}}$ with the initial simplex d with precision ± 0.004. The computational time spent in the latter case was 0.110 seconds.

Function 11 Four distinct global minima (where $f(x_*) = -1$) were found with the initial simplexes a, b, c, d with precision ± 0.0001. The computational time spent in minimization was on average 0.238 seconds. The minimization route started with initial simplex a is shown in Figure 6.

Function 12 The global minimum $x_* = 0$ was found with initial simplexes d and e with precision ± 0.006. Two distinct local minimizers were found with the initial simplexes a and c. Routine failed with the initial simplex b (division by zero). The computational time spent in minimizing was on average 0.151 seconds.

Function 13 The global minimum $x_* \simeq 0.5005$ was found with all the initial simplexes a, b, c with precision ± 0.0033. The computational time spent in minimizing was on average 0.367 seconds. The minimization route started with initial simplex a is shown in Figure 7. Figure 8 shows the graph of the function together with the route.

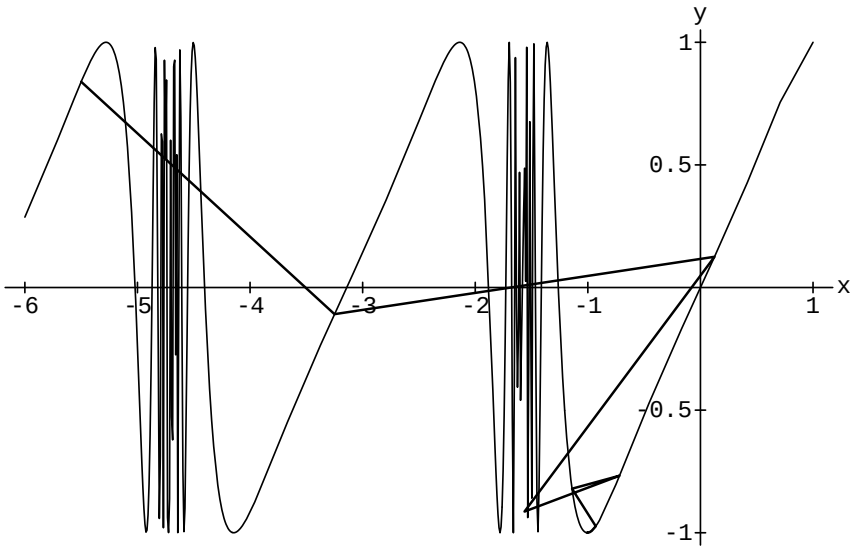

Figure 6: The minimization route of routine *Amoeba* with problem 11.a, where $f(x) = \sin\tan x$, consists of 11 routepoints. The minimum is found at $x_* \simeq -1.0044$

Function 14 Routine returned incorrect minimum points $x_* = 2.5$ and $x_* = 1.5$ with the initial simplexes a and b, respectively, but found the global minimum $x_* \simeq 0.9435$ with the initial simplex c with precision ± 0.0018. The computational time spent in the latter case was 0.165 seconds.

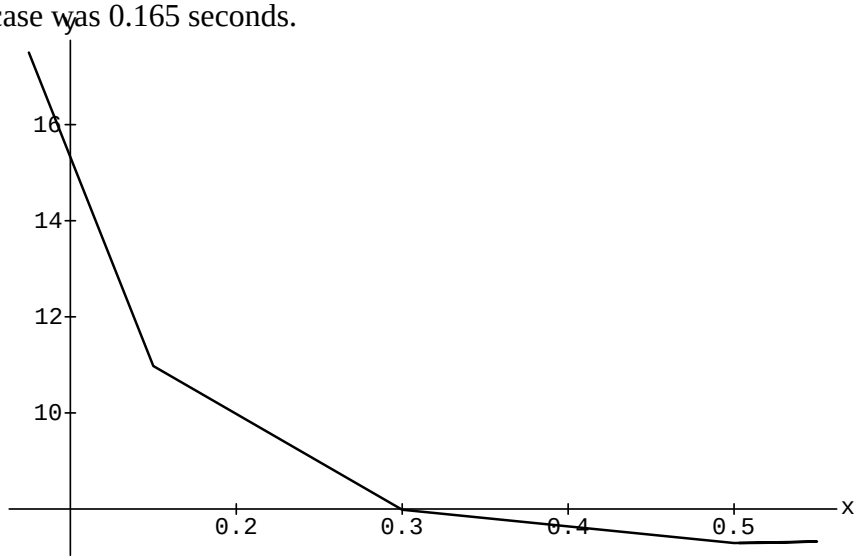

Figure 7: The minimization route of routine *Amoeba* with problem 13.a consists of 9 routepoints. The minimum is found at $x_* \simeq 0.5031$

Testing of routine *Powell* produced the following results on one-dimensional functions 10..14 (with the initial values defined in section 4.2.2) :

Function 10 Routine failed with the initial values a and c (floating-point overflow), but found the local minimum $x_* = \sqrt{\frac{2}{3}}$ with the initial value b with precision ± 0.0001. The computational time spent in the latter case was 0.495 seconds.

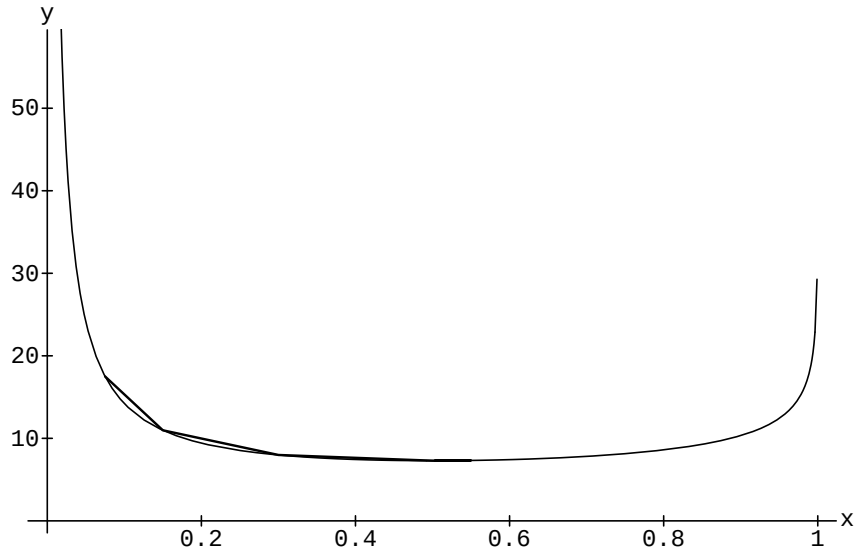

Figure 8: The minimization route of routine *Amoeba* (problem 13.a) with the graph of function $f(x) = \frac{1}{x} + K(x) + K^2(x)$

Function 11 Three distinct global minima (where $f(x_*) = -1$) were found with the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.861 seconds.

Function 12 The global minimum $x_* = 0$ was found with initial values d and e with precision ± 0.0003. Two distinct local minimizers were found with the initial values a and c. Routine failed with the initial value b (division by zero). The computational time spent in minimizing was on average 0.549 seconds.

Function 13 Routine failed with all the initial values a, b, c (domain error in sqrt).

Function 14 The global minimum $x_* \simeq 0.9435$ was found with initial value a and the local minimum $x_* \simeq 4.6010$ with initial values b and c ; all with precision ± 0.0001. The computational time spent in minimization was on average 0.861 seconds.

Testing of routine *Vmmin* produced the following results on one-dimensional functions 10..14 (with the initial values defined in section 4.2.2) :

Function 10 Routine failed with the initial values a and c (floating-point overflow), but found the local minimum $x_* = \sqrt{\frac{2}{3}}$ with the initial value b with precision ± 0.0001. The computational time spent in the latter case was 0.110 seconds.

Function 11 Three distinct global minima (where $f(x_*) = -1$) were found with the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.147 seconds.

Function 12 The global minimum $x_* = 0$ was not found. Two distinct local minimizers were found with the initial values a and c. Routine failed with the initial value b

(division by zero) and values d and e (time-out). The computational time spent in minimizing was on average 0.358 seconds.

Function 13 *Mathematica* could not generate the definition of the gradient.

Function 14 The global minimum $x_* \simeq 0.9435$ was found with initial value a and the local minimum $x_* \simeq 4.6010$ with initial values b and c ; all with precision ± 0.0001. The computational time spent in minimization was on average 0.183 seconds. The minimization route started with initial value b is shown in Figure 9.

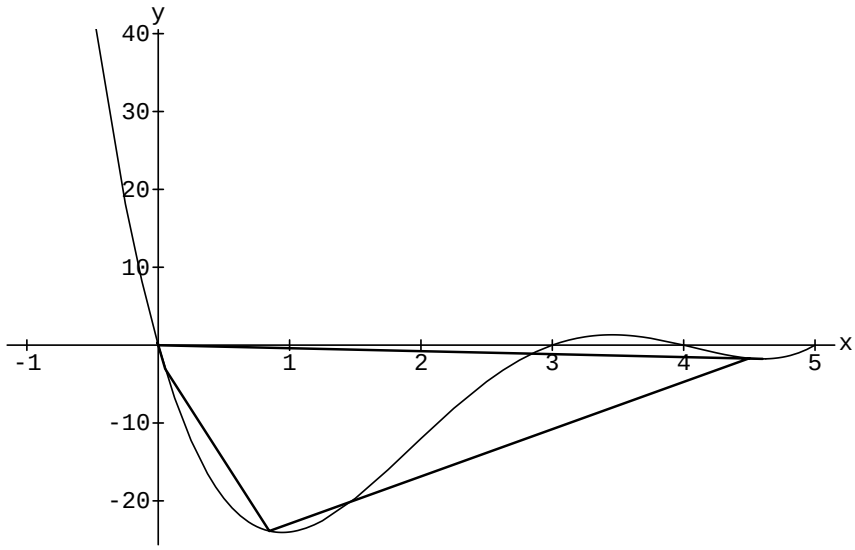

Figure 9: The minimization route of routine *Vmmin* with problem 14.b, where $f(x) = x^4 - 12x^3 + 47x^2 - 60x$, consists of 7 routepoints. The search of the minimum $x_* \simeq 4.6010$ is started at $x = 4.5$

The complete results of this test-case can be found in files AMOEBA1.STA, POWELL1.STA and VMMIN1.STA in the installation diskette.

4.3.3 Results of testing the multidimensional methods with multidimensional functions

Testing of routine *Amoeba* produced the following results on multidimensional functions 20..32 (with the initial simplexes defined in section 4.2.3) :

Function 20 The global minimum $\mathbf{x}_* = (2, -1)$ was found with initial simplex a with precision ± 0.0001. The computational time spent in minimization was 1.484 seconds. Note, that from now on, the precision of the found minimum means the precision of the least accurate coordinate. A three-dimensional plot of function number 20 is shown in Figure 17 in Appendix L.

Function 21 The global minimum $\mathbf{x}_* = (1, 1)$ was found with all the initial simplexes a, b, c with precision ± 0.0001. The computational time spent in minimization was

on average 16.245 seconds. The minimization route started with initial simplex b is shown in Figure 10. A three-dimensional plot of function number 21 is shown in Figure 19 in Appendix L.

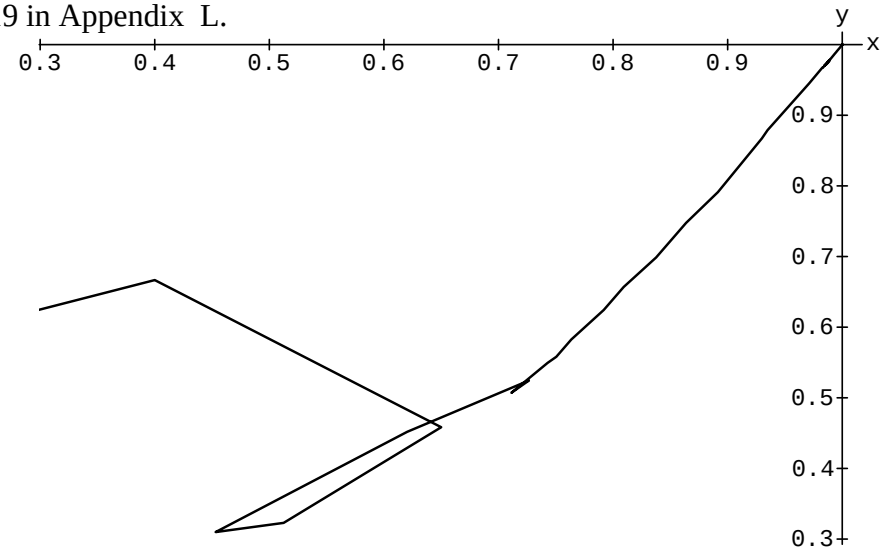

Figure 10: The minimization route of routine *Amoeba* with problem 21.b, where $f(x, y) = 100(x^2 - y)^2 + (1 - x)^2$, consists of 65 routepoints. The minimum is found at $\mathbf{x}_* = (1.0000, 1.0000)$

Function 22 The global minimum $\mathbf{x}_* = (\frac{1}{100}, 100)$ was found with initial simplex a with precision ± 0.0001. The computational time spent in minimization was 47.143 seconds. A three-dimensional plot of function number 22 is shown in Figure 21 in Appendix L.

Function 23 Routine failed with the initial simplex a (floating-point overflow). A three-dimensional plot of function number 23 is shown in Figure 23 in Appendix L.

Function 24 A global minimum $\mathbf{x}_* = (0.0, 2.5)$ was found with the initial simplex a with precision ± 0.0001. The computational time spent in minimization was 0.769 seconds. A three-dimensional plot of function number 24 is shown in Figure 25 in Appendix L.

Function 25 The global minimum $\mathbf{x}_* = (0, 0)$ was found with all the initial simplexes a and b with precision ± 0.0001. The computational time spent in minimization was on average 2.692 seconds.

Function 30 The global minimum $\mathbf{x}_* = (1, 1, 1)$ was found with all the initial simplexes a and b with precision ± 0.0001. The computational time spent in minimization was on average 31.978 seconds.

Function 31 The global minimum $\mathbf{x}_* = (1, 2, -5)$ was found with the initial simplex a with precision ± 0.006. The computational time spent in minimization was 0.989 seconds.

Function 32 Routine returned an incorrect minimum point $\mathbf{x}_* = (0.25, 0.25, 0.25)$ with the initial simplex a, but found the global minimum $\mathbf{x}_* = (0.5, 0.5, 0.5)$ with the initial simplex b with precision ± 0.0001. The computational time spent in the latter case was 3.846 seconds.

Testing of routine *Powell* produced the following results on multidimensional functions 20..32 (with the initial values defined in section 4.2.3) :

Function 20 The global minimum $\mathbf{x}_* = (2, -1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 1.465 seconds.

Function 21 The global minimum $\mathbf{x}_* = (1, 1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 12.234 seconds. The minimization route started with initial value a is shown in Figure 11.

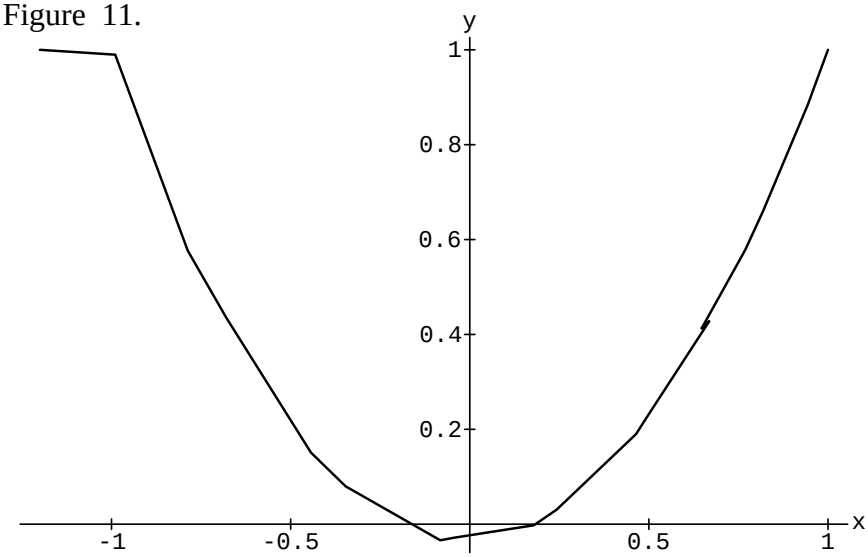

Figure 11: The minimization route of routine *Powell* with problem 21.a, where $f(x, y) = 100(x^2 - y)^2 + (1 - x)^2$, consists of 25 routepoints. The minimum is found at $\mathbf{x}_* = (1.0000, 1.0000)$

Function 22 The global minimum $\mathbf{x}_* = (\frac{1}{100}, 100)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 13.791 seconds.

Function 23 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 24 Routine returned an incorrect minimum point $\mathbf{x}_* = (-0.5046, 0.0002)$ with the initial value d, but found three distinct global minimizers (where $f(\mathbf{x}_*) = 0$) with initial values a, b, c with precision ± 0.0001. The computational time spent in the latter case was on average 1.758 seconds.

Function 25 The global minimum $\mathbf{x}_* = (0, 0)$ was found with all the initial values a, b, c with precision ± 0.01. The computational time spent in minimization was on average 3.297 seconds. The minimization route started with initial value c is shown in Figure 12.

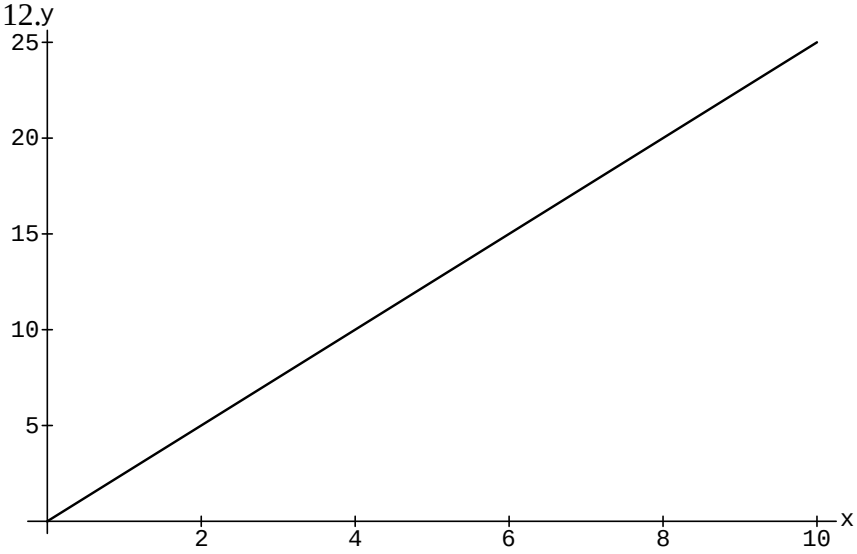

Figure 12: The minimization route of routine *Powell* with problem 25.c, where $f(r, \varphi) = r^2 + r \sin^2 \frac{\varphi}{r}$, consists of 4 routepoints. The minimum is found at $\mathbf{x}_* = (0.0018, -0.0071)$

Function 30 The global minimum $\mathbf{x}_* = (1, 1, 1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 22.674 seconds.

Function 31 The global minimum $\mathbf{x}_* = (1, 2, -5)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 1.520 seconds.

Function 32 Routine returned an incorrect minimum point $\mathbf{x}_* = (23.2595, -19.1171, 4.3319)$ with the initial value c, but found the global minimum $\mathbf{x}_* = (0.5, 0.5, 0.5)$ with the initial values a and b with precision ± 0.0001. The computational time spent in the latter case was 3.105 seconds.

Testing of routine *Vmmin* produced the following results on multidimensional functions 20..32 (with the initial values defined in section 4.2.3) :

Function 20 The global minimum $\mathbf{x}_* = (2, -1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.605 seconds.

Function 21 The global minimum $\mathbf{x}_* = (1, 1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.916 seconds. The minimization route started with initial value a is shown in Figure 13.

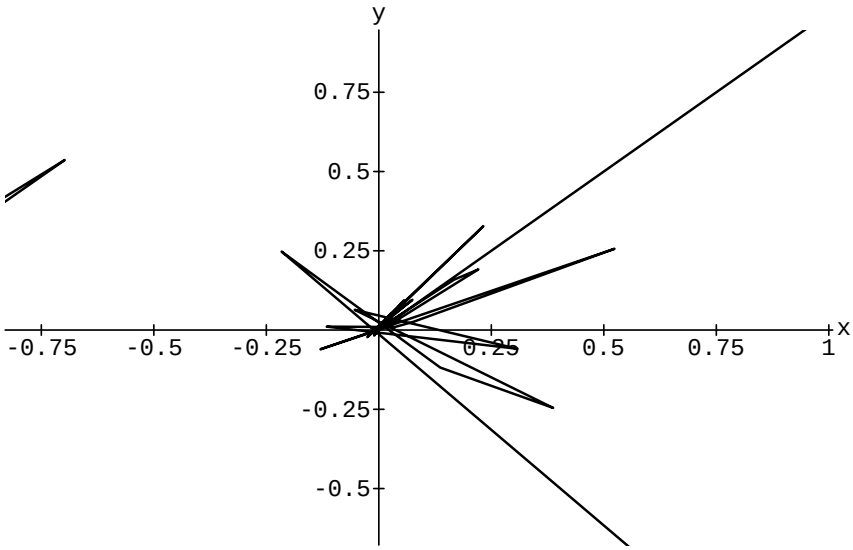

Figure 13: The minimization route of routine *Vmmin* with problem 21.a, where $f(x,y) = 100(x^2 - y)^2 + (1 - x)^2$, consists of 35 routepoints. The minimum is found at $\mathbf{x}_* = (1.0000, 1.0000)$

Function 22 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 23 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 24 Routine returned an incorrect minimum point $\mathbf{x}_* = (-9.4248, -0.0009)$ with the initial value d, but found three distinct global minimizers (where $f(\mathbf{x}_*) = 0$) with initial values a, b, c with precision ± 0.0001. The computational time spent in the latter case was on average 9.103 seconds.

Function 25 Routine failed with the initial value b (time-out), but found the global minimum $\mathbf{x}_* = (0, 0)$ with initial values a and c with precision ± 0.0017. The computational time spent in the latter case was on average 37.418 seconds.

Function 30 The global minimum $\mathbf{x}_* = (1, 1, 1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 1.502 seconds.

Function 31 The global minimum $\mathbf{x}_* = (1, 2, -5)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.147 seconds.

Function 32 The global minimum $\mathbf{x}_* = (0.5, 0.5, 0.5)$ was found with initial values a and b with precision ± 0.0001. The computational time spent in minimization was on average 1.181 seconds. A local minimum $\mathbf{x}_* = (20.1124, -15.9706, 4.3329)$ $f(\mathbf{x}_*) = 0.9858$ was found with initial value c. The computational time spent in this case was 6.703 seconds.

Testing of routine *Cgmin* produced the following results on multidimensional functions 20..32 (with the initial values defined in section 4.2.3) :

Function 20 The global minimum $\mathbf{x}_* = (2, -1)$ was found with all the initial values a, b, c with precision ± 0.0003. The computational time spent in minimization was on average 0.275 seconds.

Function 21 The global minimum $\mathbf{x}_* = (1, 1)$ was found with all the initial values a, b, c with precision ± 0.0025. The computational time spent in minimization was on average 5.458 seconds. The minimization route started with initial value a is shown in Figure 14.

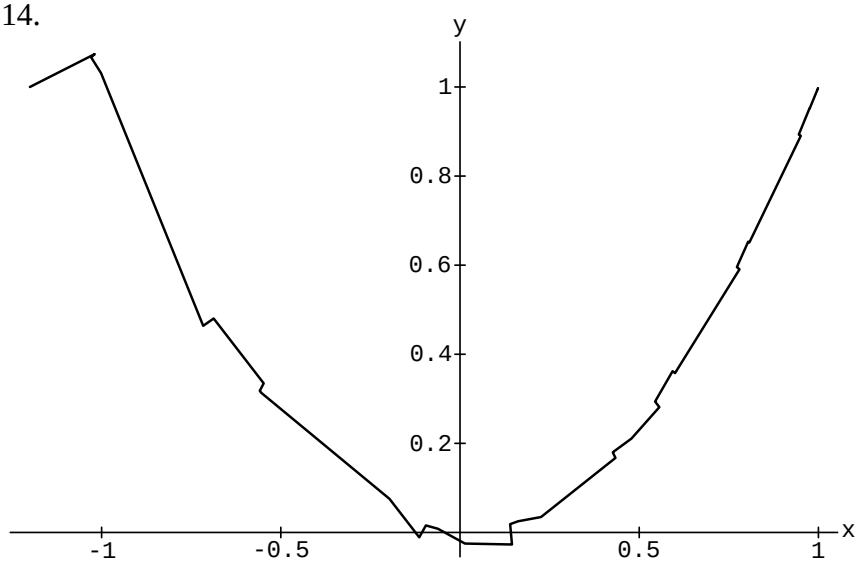

Figure 14: The minimization route of routine *Cgmin* with problem 21.a, where $f(x, y) = 100(x^2 - y)^2 + (1 - x)^2$, consists of 41 routepoints. The minimum is found at $\mathbf{x}_* = (0.9987, 0.9975)$

Function 22 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 23 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 24 Routine returned incorrect minimum points $\mathbf{x}_* = (15.7079, -10.0062)$ and $\mathbf{x}_* = (1.0000, 0.0001)$ with the initial values c and d, respectively, but found two distinct global minimizers (where $f(\mathbf{x}_*) = 0$) with initial values a and b with precision ± 0.0001. The computational time spent in the latter case was on average 0.220 seconds.

Function 25 The global minimum $\mathbf{x}_* = (0, 0)$ was found with all the initial values a, b, c with precision ± 0.0074. The computational time spent in minimization was on average 19.047 seconds. The minimization route started with initial value c is shown in Figure 15.

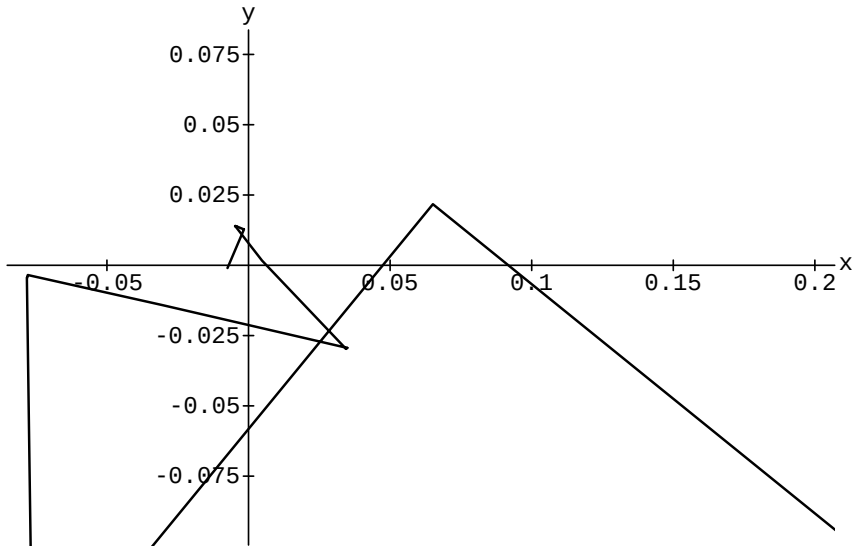

Figure 15: The minimization route of routine *Cgmin* with problem 25.c, where $f(r,\varphi) = r^2 + r\sin^2\frac{\varphi}{r}$, consists of 93 routepoints. The minimum is found at $\mathbf{x}_* = (-0.0074, -0.0010)$

Function 30 The global minimum $\mathbf{x}_* = (1,1,1)$ was found with all the initial values a, b, c with precision ± 0.0028. The computational time spent in minimization was on average 6.044 seconds.

Function 31 The global minimum $\mathbf{x}_* = (1,2,-5)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.055 seconds.

Function 32 The global minimum $\mathbf{x}_* = (0.5, 0.5, 0.5)$ was found with initial values a and b with precision ± 0.0019. The computational time spent in minimization was on average 0.632 seconds. A local minimum $\mathbf{x}_* = (20.1018, -16.0825, 4.2935)$ $f(\mathbf{x}_*) = 1.0052$ was found with initial value c. The computational time spent in this case was 15.824 seconds.

Testing of routine *Msteepdesc* produced the following results on multidimensional functions 20..32 (with the initial values defined in section 4.2.3) :

Function 20 The global minimum $\mathbf{x}_* = (2,-1)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 2.289 seconds.

Function 21 The global minimum $\mathbf{x}_* = (1,1)$ was found with all the initial values a, b, c with precision ± 0.0004. The computational time spent in minimization was on average 7.876 seconds. The minimization route started with initial value a is shown in Figure 16.

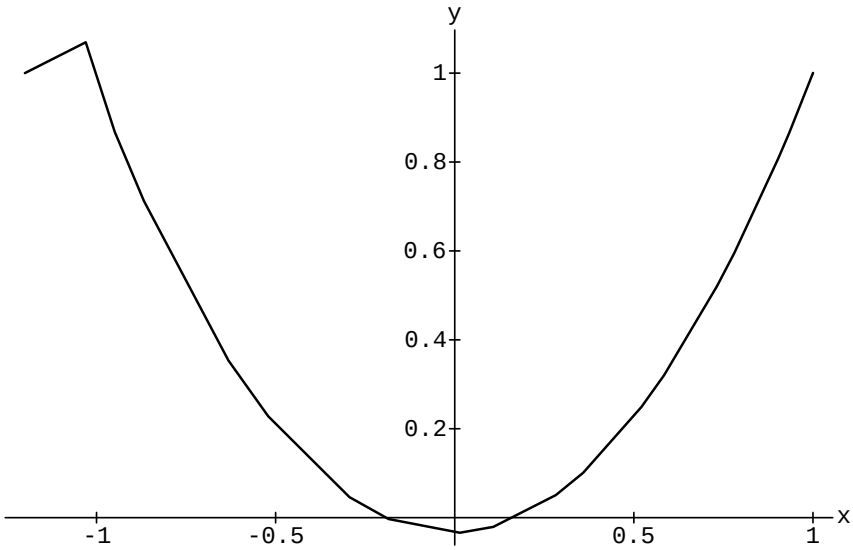

Figure 16: The minimization route of routine *Msteepdesc* with problem 21.a, where $f(x,y) = 100(x^2 - y)^2 + (1 - x)^2$, consists of 20 routepoints. The minimum is found at $\mathbf{x}_* = (1.0002, 1.0004)$

Function 22 Routine returned incorrect minimum points $\mathbf{x}_* = (0.0018, 1.0000)$, $\mathbf{x}_* = (0.0017, 0.5000)$ and $\mathbf{x}_* = (0.0016, -0.2210)$ with the initial values a, b, c, respectively.

Function 23 Routine failed with all the initial values a, b, c (floating-point overflow).

Function 24 Routine returned incorrect minimum point $\mathbf{x}_* = (15.7078, -10.0051)$ with the initial value c, but found three distinct global minimizers (where $f(\mathbf{x}_*) = 0$) with initial values a, b, d with precision ± 0.0001. The computational time spent in the latter case was on average 0.421 seconds.

Function 25 The global minimum $\mathbf{x}_* = (0, 0)$ was found with all the initial values a, b, c with precision ± 0.05. The computational time spent in minimization was on average 7.985 seconds.

Function 30 The global minimum $\mathbf{x}_* = (1, 1, 1)$ was found with all the initial values a, b, c with precision ± 0.0006. The computational time spent in minimization was on average 35.257 seconds.

Function 31 The global minimum $\mathbf{x}_* = (1, 2, -5)$ was found with all the initial values a, b, c with precision ± 0.0001. The computational time spent in minimization was on average 0.165 seconds.

Function 32 The global minimum $\mathbf{x}_* = (0.5, 0.5, 0.5)$ was found with initial values a and b with precision ± 0.0003. The computational time spent in minimization was on average 1.209 seconds. A local minimum $\mathbf{x}_* = (20.1157, -15.9708, 4.3318)$ $f(\mathbf{x}_*) = 0.9858$ was found with initial value c. The computational time spent in this case was 3.077 seconds.

The complete results of this test-case can be found in files AMOEBAN.STA, POWELLN.STA, VMMINN.STA, CGMINN.STA and STDESCN.STA in the installation diskette.

4.4 Analyzing the test results

The following three sections discuss the results of each test-case (described in sections 4.3.1, 4.3.2 and 4.3.3).

4.4.1 One-dimensional method

Brent's method proved to converge to the minimum very rapidly after the initial bracketing was done succesfully. The precision of the found minimum was in every case at least ± 0.0003. Although the CPU-time used by routine *Brent* was not mentioned in the test results, it was at most 1.0 seconds with all the initial values. The differences in performance of the bracketing routines were noticeable : in almost every case the CPU-time used by routine *Mnbrak1* was at least 50 % of the total elapsed time, while routine *Mnbrakn* used only 10-20 % of the total time. The total time with bracketing done by *Mnbrak1* was also remarkably higher than with *Mnbrakn*. Despite of the fact that *Mnbrakn* failed with the cubic function (number 10) and *Mnbrak1* succeeded in bracketing it, no conclusion can be drawn for *Mnbrak1* being superior with all such problems.

4.4.2 Multidimensional methods with one-dimensional functions

Routines *Amoeba*,*Powell* and *Vmmin* had all equal difficulties with the cubic function (number 10) and initial values a and c, which were not "near" the minimum, causing the routines to exceed the floating-point range. The minima which these routines actually found for this function were almost equally accurate.

For the rest of the functions, routine *Amoeba* proved to be the only routine which found a minimum for each of them. Both the routines *Powell* and *Vmmin* failed with the function number 13 and *Vmmin*, in addition, could not find the global minimum for function number 12. Although *Powell* seemed to converge slightly slower than the other two routines on average, *Vmmin* indicated the best accuracy and fastest convergence, and *Amoeba* proved to be robust, very little can be said about their effectiveness, especially when the elapsed CPU-time is less than 1.0 seconds.

When comparing the results of this test-case with those of *Brent* and *Mnbrakn*, no remarkable differences can be found either in accuracy or in elapsed CPU-time. Thus the multidimensional methods tested here seem to be suitable also for one-dimensional minimization.

4.4.3 Multidimensional methods with multidimensional functions

All the multidimensional methods failed with function 23. The correct answer was achieved by using *Mathematica*, e.g. with initial value $\mathbf{p} = (2.0, 3.0)$. Routines *Amoeba*

and *Powell* were the only routines which did not fail with function 22.

Because the initial values (simplexes) for routine *Amoeba* were not compatible with the other routines, its results cannot be fully compared. However, *Amoeba* proved to be quite efficient with some problems (e.g. 24, 25 and 31) and indicated fair accuracy in finding minimum with all problems.

The rest of the routines found the minima with almost equal precision (except with function 25) and thus the only way to compare their performance is by the consumed CPU-time.

Routine *Cgmin* proved to be the fastest with function number 20, routine *Vmmin* being the second fastest, but indicating slightly better accuracy than *Cgmin*. Routine *Msteepdesc* was almost ten times slower than *Cgmin* on average.

Routine *Vmmin* was the fastest with function number 21, *Cgmin* being the second fastest, but used almost five times more CPU-time than *Vmmin*. Routine *Powell* proved to be two times slower than *Msteepdesc* which was third fastest.

With function number 24 routine *Vmmin* was the slowest, using in some cases ten times more CPU-time than *Cgmin* and *Msteepdesc*, which converged to a minimum very rapidly.

With function number 25 routines *Vmmin*,*Cgmin* and *Msteepdesc* (which make use of gradient) used noticeably more CPU-time than routines *Amoeba* and *Powell*. The reason for this is the heavy computing of the numeric gradient in the external file : routine *Msteepdesc* proved to converge fastest, but with poor accuracy. Even *Powell* which does not use gradient, reported quite an inaccurate minimum. This may be a result of problems during the line minimization, that is, during the execution of routine *Linmin* which is used by *Powell* and *Msteepdesc*. Routine *Vmmin* was especially ineffiecient : in one case it run out of its allowed time limit and in other cases it used five times more CPU-time than *Msteepdesc*.

Routines *Vmmin* and *Cgmin* were clearly the most efficient in minimizing the functions 30..32. Routine *Msteepdesc* converged faster than *Powell* with functions 31 and 32, but used 35 times more CPU-time with function 30 than *Vmmin* on average.

As a conclusion of results of this test-case it can be said that variable metric and conjugate gradient algorithms are the most effective choices for solving an average problem, whenever the computation of function's gradient is possible.

A Bracketing a minimum – routine MNBRAK1

```
/* *************** */
/* Module MNBRAK1.C */
/* *************** */

#include <math.h>
#include "nrutil.h"

#define GOLD 1.618034 /* Magnification ratios by which successive intervals
                         are magnified : the first is the default ratio */
#define MAXMG 50      /* and this is the maximum magnification.         */
#define ITER1 50      /* Maximum numbers of iterations in inner loops   */
#define ITER2 100
#define MIN(a,b) ((a) < (b) ? (a) : (b))
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

/* ************************************** */
   void mnbrak(ax,bx,cx,fa,fb,fc,func)
/* ************************************** */
/* Routine for bracketing a minimum in one-
   dimension by author of this study          */
/* ************************************** */
float *ax,*bx,*cx,*fa,*fb,*fc;
float (*func)();
{
 float starta,startb,dum,a,b,c,dx,factor;
 int counter,j,times,found;

 starta = *ax;
 startb = *bx;

 /* b should be greater than a */
 if (starta > startb) {
    SHFT(dum,starta,startb,dum)
 }
 dx = startb - starta;
 factor = GOLD;         /* Use default magnification */
 times = 1;
 found = 0;

 /* Do the next loop until we bracket (two times at most) or until
    too much CPU-time is used  */
 while ((!found) && (times <= 2) && (!timeout())) {
    a = starta;
    b = startb;
    counter = 0;

    /* Do the next loop until we bracket (ITER1 times at most) or until
       too much CPU-time is used  */
    while ((!found) && (counter <= ITER1) && (!timeout())) {
          j = 1;
          c = (a+b)/2;  /* First guess for c */
          /* Search between a and b */
          while ((j<ITER2) && (MIN((*func)(a), (*func)(b)) < (*func)(c))) {
                c = a + (j * (b-a)/ITER2);
                j++;
          }
          found = ((*func)(c) < MIN((*func)(a),(*func)(b)));
          if (!found) {
             /* No success. Now we go downhill ... */
             if ((*func)(a) < (*func)(b)) {
                 /* Downhill direction is from b to a */
                 b = (a+b)/2;
                 a = a - (factor * dx);
```

```
            }
            else {
                /* Downhill direction is from a to b */
                a = (a+b)/2;
                b = b + (factor * dx);
            }
            counter++;
        }
    }

    if (!found) {
        /* No success. Restore original values for a and b */
        a = starta;
        b = startb;
        counter = 0;
    }

    /* Do the next loop until we bracket (ITER1 times at most) or until
       too much CPU-time is used  */
    while ((!found) && (counter <= ITER1) && (!timeout())) {
          j = 1;
          c = (a+b)/2;  /* First guess for c */
          /* Search between a and b */
          while ((j<ITER2) && (MIN((*func)(a), (*func)(b)) < (*func)(c))) {
                c = a + (j * (b-a)/ITER2);
                j++;
          }
          found = ((*func)(c) < MIN((*func)(a),(*func)(b)));
          if (!found) {
             /* No success. Now we go uphill ... */
             if ((*func)(a) > (*func)(b)) {
                 /* Uphill direction is from b to a */
                 b = (a+b)/2;
                 a = a - (factor * dx);
             }
             else {
                 /* Uphill direction is from a to b */
                 a = (a+b)/2;
                 b = b + (factor * dx);
             }
             counter++;
          }
    }

    if (!found) {
       /* No success. Use maximum magnification */
       factor = MAXMG;
       times++;
    }
 }
 /* Success or not, return the triplet with function values */
 *ax = a;
 *bx = c;
 *cx = b;
 *fa = (*func)(a);
 *fb = (*func)(c);
 *fc = (*func)(b);
}

#undef GOLD
#undef MAXMG
#undef MIN
#undef SHFT
```

B Bracketing a minimum – routine MNBRAKN

```
/* ***************** */
/* Module MNBRAKN.C */
/* ***************** */

#include <math.h>
#include "nrutil.h"

#define GOLD 1.618034 /* Default ratio by which successive intervals are
                         magnified */
#define GLIMIT 100.0  /* Maximum magnification allowed for a parabolic-fit
                         step */
#define TINY 1.0e-20  /* Used to prevent any possible divisions by zero */
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

/* Returns TRUE, if (a-b)(c-d) > 0 , otherwise FALSE */
int compare(a,b,c,d)
float a,b,c,d;
{
  if (((a > b) && (c > d)) ||
     ((a < b) && (c < d)))
     return(TRUE);
  else
     return(FALSE);
}

/* Returns TRUE, if (a-b)(c-d) >= 0 , otherwise FALSE */
int eqcompare(a,b,c,d)
float a,b,c,d;
{
  if (((a > b) && (c > d)) ||
     ((a < b) && (c < d)) ||
     (a == b) ||
     (c == d))
     return(TRUE);
  else
     return(FALSE);
}

/* ************************************** */
   void mnbrak(ax,bx,cx,fa,fb,fc,func)
/* ************************************** */
/* Routine for bracketing a minimum in one-
   dimension by Press et al                */
/* ************************************** */
float *ax,*bx,*cx,*fa,*fb,*fc;
float (*func)();
{
 float ulim,u,r,q,fu,dum;

 *fa=(*func)(*ax);
 *fb=(*func)(*bx);
 if (*fb > *fa) {
    /* Switch roles of a and b so that we can go downhill in the
       direction from a to b */
    SHFT(dum,*ax,*bx,dum)
    SHFT(dum,*fb,*fa,dum)
 }
 *cx=(*bx)+GOLD*(*bx-*ax);  /* First guess for c */
 *fc=(*func)(*cx);

 /* Do the next loop until we bracket or until too much CPU-time is used */
```

```
 while ((*fb > *fc) && (!timeout())) {
        r=(*bx-*ax)*(*fb-*fc);
        q=(*bx-*cx)*(*fb-*fa);
        /* Compute u by parabolic extrapolation from a,b and c */
        u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/
                (2.0*SIGN(MAX(fabs(q-r),TINY),q-r));
        ulim=(*bx)+GLIMIT*(*cx-*bx); /* We won't go farther than this */
        if (compare(*bx,u,u,*cx)) {
           /* Parabolic u is between b and c, so try it */
           fu=(*func)(u);
           if (fu < *fc) {
              /* Got a minimum between b and c */
              *ax=(*bx);
              *bx=u;
              *fa=(*fb);
              *fb=fu;
              return; /* Exit routine */
           } else if (fu > *fb) {
                     /* Got a minimum between a and u */
                     *cx=u;
                     *fc=fu;
                     return; /* Exit routine */
                  }
           /* Parabolic fit was no use. Use default magnification */
           u=(*cx)+GOLD*(*cx-*bx);
           fu=(*func)(u);
        } else if (compare(*cx,u,u,ulim)) {
                  /* Parabolic fit is between c and its allowed limit */
                  fu=(*func)(u);
                  if (fu < *fc) {
                     SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                     SHFT(*fb,*fc,fu,(*func)(u))
                  }
               } else if (eqcompare(u,ulim,ulim,*cx)) {
                         /* Limit parabolic u to maximum allowed value */
                         u=ulim;
                         fu=(*func)(u);
                       } else {
                              /* Reject parabolic u. Use default
                                 magnification */
                              u=(*cx)+GOLD*(*cx-*bx);
                              fu=(*func)(u);
                         }
        /* Eliminate the oldest point and continue */
        SHFT(*ax,*bx,*cx,u)
        SHFT(*fa,*fb,*fc,fu)
 }
}

#undef GOLD
#undef GLIMIT
#undef TINY
#undef MAX
#undef SIGN
#undef SHFT
```

C Brent's method – routine BRENT

```
/* ************** */
/* Module BRENT.C */
/* ************** */

#include <math.h>
#include "nrutil.h"
```

```
#define ITMAX 500       /* Maximum number of iterations */
#define CGOLD 0.3819660 /* Golden ratio */
#define ZEPS 1.0e-10    /* Used to prevent round-off errors     */
#define SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

extern float *routearr; /* External variables ; defined in driver module */
extern int routei;
extern int powellflag;

/* ******************************** */
   float brent(ax,bx,cx,f,tol,xmin)
/* ******************************** */
/* Brent's method in one-dimension
   (adapted from Press et al)        */
/* ******************************** */
float ax,bx,cx,tol,*xmin;
float (*f)();
{
 int iter,dummy;
 float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
 float e=0.0;

 a=((ax < cx) ? ax : cx); /* Initializations ... */
 b=((ax > cx) ? ax : cx);
 x=w=v=bx;
 fw=fv=fx=(*f)(x);

 /* Route saving begin */
 if ((!powellflag) && (routei <= MAXROUTEPTS))
    routearr[routei++] = x;
 /* Route saving end   */

 /* Main loop */
 for (iter=1;iter<=ITMAX;iter++) {
         dummy = timeout();  /* if too much CPU-time is used then halt */
         xm=0.5*(a+b);
         tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
         if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
            /* Minimum was found */
            *xmin=x;
            return fx;
         }
         if (fabs(e) > tol1) {
             /* Construct a trial parabolic fit */
             r=(x-w)*(fx-fv);
             q=(x-v)*(fx-fw);
             p=(x-v)*q-(x-w)*r;
             q=2.0*(q-r);
             if (q > 0.0) p = -p;
             q=fabs(q);
             etemp=e;
             e=d;
             /* Determine the acceptability of the parabolic fit */
             if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) ||
                 p >= q*(b-x))
                 /* Take the golden section step */
                 d=CGOLD*(e=(x >= xm ? a-x : b-x));
             else {
                    /* Take the parabolic step */
                    d=p/q;
                    u=x+d;
                    if (u-a < tol2 || b-u < tol2)
                        d=SIGN(tol1,xm-x);
```

```
            }
        } else {
                /* Take the golden section step */
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
          }
        u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
        fu=(*f)(u);
        /* Do the housekeeping */
        if (fu <= fx) {
                if (u >= x) a=x; else b=x;
                SHFT(v,w,x,u)
                SHFT(fv,fw,fx,fu)
                /* Route saving begin */
                if ((!powellflag) && (routei <= MAXROUTEPTS))
                   routearr[routei++] = x;
                /* Route saving end   */
        } else {
                if (u < x) a=u; else b=u;
                if (fu <= fw || w == x) {
                        v=w;
                        w=u;
                        fv=fw;
                        fw=fu;
                } else if (fu <= fv || v == x || v == w) {
                        v=u;
                        fv=fu;
                }
        }
 }
 /* Too much iterations, but return the best point anyway */
 *xmin=x;
 return fx;
}

#undef ITMAX
#undef CGOLD
#undef ZEPS
#undef SIGN
```

D Downhill simplex method – routine AMOEBA

```
/* *************** */
/* Module AMOEBA.C */
/* *************** */

#include <math.h>
#include "nrutil.h"

#define NMAX 5000 /* Maximum number of iterations */
#define ALPHA 1.0 /* Constants defining the expansions and contractions */
#define BETA 0.5
#define GAMMA 2.0

#define GET_PSUM for (j=1;j<=ndim;j++) { for (i=1,sum=0.0;i<=mpts;i++)\
                                                 sum += p[i][j]; psum[j]=sum;}

extern float **routearr; /* External variables ; defined in driver module */
extern int routei;

/* ************************************************* */
   float amotry(p,y,psum,ndim,funk,ihi,nfunk,fac)
/* ************************************************* */
/* Subroutine for doing extrapolations and housekeeping */
/* ************************************************* */
```

```
float **p,*y,*psum,(*funk)(),fac;
int ndim,ihi,*nfunk;
{
 int j;
 float fac1,fac2,ytry,*ptry;

 ptry=vector(ndim);
 fac1=(1.0-fac)/ndim;
 fac2=fac1-fac;
 for (j=1;j<=ndim;j++) ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
 ytry=(*funk)(ptry);
 ++(*nfunk);
 if (ytry < y[ihi]) {
         y[ihi]=ytry;
         for (j=1;j<=ndim;j++) {
                 psum[j] += ptry[j]-p[ihi][j];
                 p[ihi][j]=ptry[j];
         }
 }
 free_vector(ptry);
 return ytry;
}

/* ******************************************************** */
   void amoeba(p,y,ndim,ftol,funk,nfunk)
/* ******************************************************** */
/* Downhill simplex method in multidimensions by Nelder and
   Mead (adapted from Press et al)                          */
/* ******************************************************** */
float **p,*y,ftol,(*funk)();
int ndim,*nfunk;
{
 int i,j,ilo,ihi,inhi,mpts=ndim+1,dummy;
 float ytry,ysave,sum,rtol,*psum,*tmparr,tmpdiv;

 psum = vector(ndim);
 tmparr = vector(ndim);
 *nfunk = 0;
 GET_PSUM
 /* Route saving begin */
 if (routei <= MAXROUTEPTS) {
    for (j=1;j<=ndim;j++)
        tmparr[j] = 0.0; /* Initialize temporary array */

    /* Evaluate the mean value of the vertices */
    for (j=1;j<=ndim;j++)
        for (i=1;i<=mpts;i++)
             tmparr[j] = tmparr[j] + (p[i][j] / mpts);

    for (j=1;j<=ndim;j++)
        /* Save this as a routepoint */
        routearr[routei][j] = tmparr[j];

    routei++; /* Increase the counter */
 }
 /* Route saving end */

 /* Main loop */
 for (;;) {
         dummy = timeout(); /* if too much CPU-time used then halt */
         ilo=1;
         /* Determine which point is the highest (worst), next-highest
            and lowest(best) i.e. indexes ihi,inhi and ilo */
         ihi = y[1]>y[2] ? (inhi=2,1) : (inhi=1,2);
         for (i=1;i<=mpts;i++) {
```

```
        if (y[i] < y[ilo]) ilo=i;
        if (y[i] > y[ihi]) {
                inhi=ihi;
                ihi=i;
        } else if (y[i] > y[inhi])
                if (i != ihi) inhi=i;
}
tmpdiv = fabs(y[ihi])+fabs(y[ilo]);
if (tmpdiv < 1.0e-11) {
   tmpdiv = 1.0e-11;   /* Prevent dividing by zero */
}
/* Compute the fractional range from highest to lowest and
   return if satisfactory. Also return if too much iterations. */
rtol=2.0*fabs(y[ihi]-y[ilo])/tmpdiv;
if ((rtol < ftol) || (*nfunk >= NMAX))
   /* Exit the loop and the routine */
   break;

/* During every iteration we will explore along the ray
   from the high point through the center of the face
   across from the high point */

/* Extrapolate by a factor ALPHA through the face, i.e.
   reflect the simplex from the high point */
ytry=amotry(p,y,psum,ndim,funk,ihi,nfunk,-ALPHA);
if (ytry <= y[ilo])
        /* Had a result better than the best point, so
           try additional extrapolation by a factor GAMMA */
        ytry=amotry(p,y,psum,ndim,funk,ihi,nfunk,GAMMA);
else if (ytry >= y[inhi]) {
        /* The reflected point is worse than than the second-
           highest. Now look for an intermediate lower point,
           in other words, perform a contraction of the
           simplex along one dimension */
        ysave=y[ihi]; /* Save the current highest point */
        ytry=amotry(p,y,psum,ndim,funk,ihi,nfunk,BETA);
        if (ytry >= ysave) {
                /* Contraction gave no improvement. Contract
                   now around the lowest (best) point */
                for (i=1;i<=mpts;i++) {
                    if (i != ilo) {
                        for (j=1;j<=ndim;j++) {
                             psum[j]=0.5*(p[i][j]+p[ilo][j]);
                             p[i][j]=psum[j];
                        }
                        y[i]=(*funk)(psum);
                    }
                }
                *nfunk += ndim;
                GET_PSUM
        }
}
/* Route saving begin */
if (routei <= MAXROUTEPTS) {
   for (j=1;j<=ndim;j++)
       tmparr[j] = 0.0; /* Initialize temporary array */

   /* Evaluate the mean value of the vertices */
   for (j=1;j<=ndim;j++)
       for (i=1;i<=mpts;i++)
            tmparr[j] = tmparr[j] + (p[i][j] / mpts);

   for (j=1;j<=ndim;j++)
       /* Save this as a routepoint */
       routearr[routei][j] = tmparr[j];
```

```
            routei++; /* Increase the counter */
          }
          /* Route saving end */
 }
 free_vector(psum);
 free_vector(tmparr);
}

#undef ALPHA
#undef BETA
#undef GAMMA
#undef NMAX
```

E Line minimization – routine F1DIM

```
/* ************** */
/* Module F1DIM.C */
/* ************** */

#include "nrutil.h"

extern int ncom;        /* Defined in linmin.c */
extern float *pcom,*xicom,(*nrfunc)();

/* ***************************************** */
   float f1dim(x)
/* ***************************************** */
/* Subroutine called by linmin               */
/* ***************************************** */
float x;
{
 int j;
 float f,*xt;

 xt=vector(ncom);
 for (j=1;j<=ncom;j++)
     xt[j]=pcom[j]+x*xicom[j];
 f=(*nrfunc)(xt);
 free_vector(xt);
 return f;
}
```

F Line minimization – routine LINMIN

```
/* *************** */
/* Module LINMIN.C */
/* *************** */

#include "nrutil.h"
#include "routines.h"

#define TOL 2.0e-4

int ncom=0;
float *pcom=0,*xicom=0,(*nrfunc)();

/* ***************************************** */
   void linmin(p,xi,n,fret,func)
/* ***************************************** */
/* Line minimization routine                 */
/* ***************************************** */
float *p,*xi,*fret,(*func)();
```

```
int n;
{
 int j;
 float xx,xmin,fx,fb,fa,bx,ax;

 ncom=n;
 pcom=vector(n);
 xicom=vector(n);
 nrfunc=func;
 for (j=1;j<=n;j++) {
      pcom[j]=p[j];
      xicom[j]=xi[j];
 }
 ax=0.0;
 xx=1.0;
 bx=2.0;
 mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
 *fret=brent(ax,xx,bx,f1dim,TOL,&xmin);
 for (j=1;j<=n;j++) {
      xi[j] *= xmin;
      p[j] += xi[j];
 }
 free_vector(xicom);
 free_vector(pcom);
}

#undef TOL
```

G Direction set method – routine POWELL

```
/* *************** */
/* Module POWELL.C */
/* *************** */

#include <math.h>
#include "nrutil.h"
#include "routines.h"

#define ITMAX 500   /* Maximum number of iterations */
static float sqrarg;
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)

extern float **routearr; /* External variables ; defined in driver module */
extern int routei;

/* ************************************************ */
   void powell(p,xi,n,ftol,iter,fret,func)
/* ************************************************ */
/* Direction set (Powell's) method in multidimensions
   (adapted from Press et al)                       */
/* ************************************************ */
float *p,**xi,ftol,*fret,(*func)();
int n,*iter;
{
 int i,ibig,j,dummy;
 float t,fptt,fp,del;
 float *pt,*ptt,*xit;

 pt=vector(n);
 ptt=vector(n);
 xit=vector(n);
 *fret=(*func)(p);
 for (j=1;j<=n;j++)
     pt[j]=p[j];        /* Save the initial point */
```

```
 /* Main loop */
 for (*iter=1;;(*iter)++) {
     dummy = timeout();         /* if too much CPU-time used then halt */
     /* Route saving begin */
     if (routei <= MAXROUTEPTS) {
        for (i=1;i<=n;i++)
            routearr[routei][i] = p[i]; /* Save this as a routepoint */
        routei++;                       /* Increase the counter      */
     }
     /* Route saving end */
     fp=(*fret);
     ibig=0;
     del=0.0;   /* Will be the biggest function decrease */
     for (i=1;i<=n;i++) {
         /* In each iteration, loop over all directions in the set */
         for (j=1;j<=n;j++)
             xit[j]=xi[j][i];   /* Copy the direction ... */
         fptt=(*fret);
         linmin(p,xit,n,fret,func); /* ... and minimize along it */
         /* ... and record it if it is the largest decrease so far */
         if (fabs(fptt-(*fret)) > del) {
            del=fabs(fptt-(*fret));
            ibig=i;
         }
     }
     if ((2.0*fabs(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))) ||
         (*iter >= ITMAX))
        /* Exit routine normally also if too much iterations */
        break;

     /* Construct the extrapolated point and the average direction moved */
     for (j=1;j<=n;j++) {
         ptt[j]=2.0*p[j]-pt[j]; /* Save the old starting point */
         xit[j]=p[j]-pt[j];
         pt[j]=p[j];
     }
     fptt=(*func)(ptt);         /* Function value at extrapolated point */
     if (fptt < fp) {
         t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)-del*SQR(fp-fptt);
         if (t < 0.0) {
            /* Use the new direction : minimize along it and save it */
            linmin(p,xit,n,fret,func);
            for (j=1;j<=n;j++)
                xi[j][ibig]=xit[j];
         }
     }
 }
 /* Route saving begin */
 if (routei <= MAXROUTEPTS) {
    for (i=1;i<=n;i++)
        routearr[routei][i] = p[i]; /* Save this as a routepoint */
    routei++;                       /* Increase the counter      */
 }
 /* Route saving end */
 free_vector(xit);
 free_vector(ptt);
 free_vector(pt);
}

#undef ITMAX
#undef SQR
```

H Variable metric method – routine VMMIN

```
/* ************** */
/* Module VMMIN.C */
/* ************** */

#include "nrutil.h"
#include "routines.h"

#define STEPREDN 0.2  /* Factor to reduce stepsize in linear search */
#define ACCTOL 1.0E-4 /* Acceptable point tolerance */

extern float **routearr; /* External variables ; defined in driver module */
extern int routei;

/* ******************************************************** */
   void vmmin(n,xvec,x,fmin,fnc,dfnc)
/* ******************************************************** */
/* Variable metric method in multidimensions by Fletcher and
   Nash (adapted from Nash)                                 */
/* ******************************************************** */
int n;
float *xvec,*x,(*fnc)(),*fmin;
void (*dfnc)();
{
  int accpoint,count,funcount,gradcount,i,j,ilast;
  float **b,*c,*g,*t;
  float d1,d2,f,gradproj,s,steplength;

  b = matrix(n,n);
  c = vector(n);
  g = vector(n);
  t = vector(n);

  f = (*fnc)(xvec);
  *fmin = f;    /* Save the best value so far */
  funcount = 1;
  gradcount = 1;
  (*dfnc)(xvec,g);
  ilast = gradcount; /* Set count to force initialization of B */

  /* Route saving begin */
  if (routei <= MAXROUTEPTS) {
     for (i=1;i<=n;i++)
         routearr[routei][i] = xvec[i]; /* Save this as a routepoint */
     routei++;                          /* Increase the counter      */
  }
  /* Route saving end */

  /* Main loop. Terminates when no progress can be made, and B is a unit
     matrix so that search is a steepest descent direction */
  do {
      if (ilast == gradcount) {
         /* Initialize B */
         for (i=1;i<=n;i++) {
             for (j=1;j<=n;j++)
                 b[i][j] = 0.0;
             b[i][i] = 1.0;
         }
      }
      /* Save best parameters and the gradient */
      for (i=1;i<=n;i++) {
          x[i] = xvec[i];
          c[i] = g[i];
      }
```

```
gradproj = 0.0;
for (i=1;i<=n;i++) {
    s = 0.0;
    for (j=1;j<=n;j++)
        s = s - (b[i][j] * g[j]);
    t[i] = s;
    gradproj = gradproj + (s * g[i]);
}
/* Test for descent direction */
if (gradproj < 0.0) {
   /* Begin linear search */
   steplength = 1.0; /* Try full step first */
   accpoint = 0;
   /* Line search loop */
   do {
       count = 0;
       for (i=1;i<=n;i++) {
           xvec[i] = x[i] + (steplength * t[i]);
           if (x[i] == xvec[i])
              count++;
       }
       /* Main convergence test */
       if (count < n) {
           /* Can proceed with linear search */
           f = (*fnc)(xvec);
           funcount++;
           accpoint = (f <= (*fmin + (gradproj * steplength * ACCTOL)));
           /* A point is acceptable if it satisfies the above
              criterion */
           if (!accpoint)
              steplength = steplength * STEPREDN;
       }
   } while ((count != n) && (!accpoint) && (!timeout()));
   /* End of loop for line search. If too much CPU-time used
      then halt */

   if (count < n) {
      *fmin = f;
      (*dfnc)(xvec,g);
      gradcount++;
      /* Prepare for matrix update */
      d1 = 0.0;
      for (i=1;i<=n;i++) {
          t[i] = steplength * t[i];
          c[i] = g[i] - c[i]; /* To compute vector y */
          d1 = d1 + (t[i] * c[i]);
      }
      /* Test if update is possible */
      if (d1 > 0) {
          /* Update ... */
          d2 = 0.0;
          for (i=1;i<=n;i++) {
              s = 0.0;
              for (j=1;j<=n;j++)
                  s = s + (b[i][j] * c[j]);
              x[i] = s;
              d2 = d2 + (s * c[i]);
          }
          d2 = 1.0 + (d2 / d1);
          for (i=1;i<=n;i++)
              for (j=1;j<=n;j++)
                  b[i][j] = b[i][j] -
                            ((t[i] * x[j]) + (x[i] * t[j]) -
                            (d2 * t[i] * t[j])) / d1;
```

```
            }
            /* Update was not possible. Force a restart with B set
               to unity */
            else ilast = gradcount;
         }
         else /* Count == n, cannot proceed */
              if (ilast < gradcount) {
                 count = 0; /* To force a steepest descent try */
                 ilast = gradcount; /* To reset to steepest desc. search */
              }  /* else routine has converged */
      }
      else {
            /* Uphill search direction */
            count = 0;
            if (ilast == gradcount)
               count = n; /* Force convergence */
            else
               ilast = gradcount; /* Force a restart with B set to unity */
      }

      /* Route saving begin */
      if (routei <= MAXROUTEPTS) {
         for (i=1;i<=n;i++)
             routearr[routei][i] = x[i]; /* Save this as a routepoint */
         routei++;                       /* Increase the counter      */
      }
      /* Route saving end */

  } while (((count != n) || (ilast != gradcount)) && (!timeout()));
    /* if too much CPU-time used then halt */

  free_matrix(b,n,n);
  free_vector(c);
  free_vector(g);
  free_vector(t);

}
```

I Conjugate gradients method – routine CGMIN

```
/* ************** */
/* Module CGMIN.C */
/* ************** */

#include <math.h>
#include "nrutil.h"
#include "routines.h"

#define STEPREDN 0.2  /* Factor to reduce stepsize in linear search */
#define ACCTOL 1.0E-4 /* Acceptable point tolerance */

extern float **routearr; /* External variables ; defined in driver module */
extern int routei;

/* ******************************************************** */
   void cgmin(n,xvec,x,intol,fmin,fnc,dfnc)
/* ******************************************************** */
/* Conjugate gradients method in multidimensions by Fletcher,
   Reeves and Nash (adapted from Nash)                      */
/* ******************************************************** */
int n;
float *xvec,*x,intol,(*fnc)(),*fmin;
void (*dfnc)();
{
```

```
int accpoint,count,cycle,cyclimit,funcount,gradcount,i;
float *c,*g,*t,f,g1,g2,g3,gradproj,tol,newstep,oldstep;
float setstep,steplength;

c = vector(n);
g = vector(n);
t = vector(n);

setstep = 1.7; /* Factor to increase the step in line search */
cyclimit = n;
tol = intol * n * sqrt(intol); /* Gradient test tolerance */
f = (*fnc)(xvec);
*fmin = f; /* Save the best value so far */
funcount = 1;
gradcount = 0;
/* Main loop. Terminates when no progress can be made, and search is a
   steepest descent */
do {
   for (i=1;i<=n;i++) {
      t[i] = 0.0;
      c[i] = 0.0;
   }
   cycle = 0;
   oldstep = 1.0;  /* Initially a full step */
   count = 0;
   /* Cg cycle loop */
   do {
       cycle++;
       gradcount++;
       (*dfnc)(xvec,g);
       g1 = 0.0;
       g2 = 0.0;
       for (i=1;i<=n;i++) {
          x[i] = xvec[i]; /* Save best parameters */

          /* Fletcher_Reeves */
          g1 = g1 + g[i] * g[i];
          g2 = g2 + c[i] * c[i];

          /* Polak_Ribiere
          g1 = g1 + g[i] * (g[i] - c[i]);
          g2 = g2 + c[i] * c[i]; */

          /* Beale_Sorenson
          g1 = g1 + g[i] * (g[i] - c[i]);
          g2 = g2 + t[i] * (g[i] - c[i]); */

          c[i] = g[i]; /* Save gradient */
       }

       /* Route saving begin */
       if (routei <= MAXROUTEPTS) {
          for (i=1;i<=n;i++)
              routearr[routei][i] = x[i]; /* Save this as a routepoint */
          routei++;                       /* Increase the counter      */
       }
       /* Route saving end */

       if (g1 > tol) {
            /* Descent sufficient to proceed ; generate direction */
            if (g2 > 0.0)
               g3 = g1 / g2;
            else
               g3 = 1.0;
            gradproj = 0.0;
```

```
            for (i=1;i<=n;i++) {
               t[i] = t[i] * g3 - g[i];
               gradproj = gradproj + t[i] * g[i];
            }
            steplength = oldstep;
            accpoint = 0;
            /* Line search loop */
            do {
                count = 0;
                for (i=1;i<=n;i++) {
                    xvec[i] = x[i] + steplength * t[i];
                    if (x[i] == xvec[i]) count++;
                }
                /* Main convergence test */
                if (count < n) {
                     /* Can proceed with linear search */
                     f = (*fnc)(xvec);
                     funcount++;
                     accpoint = (f <= *fmin + gradproj * steplength * ACCTOL);
                     /* A point is acceptable if it satisfies the above
                        criterion */
                     if (!accpoint)
                         steplength = steplength * STEPREDN;
                }
            } while ((count < n) && (!accpoint) && (!timeout()));
            /* End of loop for line search. If too much CPU-time used
               then halt */

            if (count < n) {
               newstep = 2 * ((f - *fmin) - gradproj * steplength);
               /* Quadratic inverse interpolation */
               if (newstep > 0) {
                   newstep = (- gradproj * steplength * steplength) / newstep;
                   for (i=1;i<=n;i++)
                       xvec[i] = x[i] + newstep * t[i];
                   /* Save the new lowest point */
                   *fmin = f;
                   f = (*fnc)(xvec);
                   funcount++;
                   if (f < *fmin)
                      *fmin = f;
                   else /* Reset to best xvec */
                      for (i=1;i<=n;i++)
                          xvec[i] = x[i] + steplength * t[i];
                 }
            }
         }
         oldstep = setstep * steplength;   /* A heuristic to prepare next
                                              iteration ... */
         if (oldstep > 1.0) oldstep = 1.0; /* ... with a limitation to
                                              prevent too large a step taken */

      } while ((count < n) && (g1 > tol) && (cycle < cyclimit) && (!timeout()));
        /* if too much CPU-time used then halt */

  } while (((cycle != 1) || ((count < n) && (g1 > tol))) && (!timeout()));
    /* if too much CPU-time used then halt */

  free_vector(c);
  free_vector(g);
  free_vector(t);
}
```

J Steepest descent method – routine MSTEEPDESC

```
/* *************** */
/* Module STDESC.C */
/* *************** */

#include <math.h>
#include "nrutil.h"
#include "routines.h"

#define MAXITER 500  /* Maximum number of iterations */
#define TOL1 5.0E-4  /* Gradient test tolerance */
#define TOL2 1.0E-4  /* Function decrease test tolerance */
#define TINY 1.0E-20 /* Used to prevent any possible divisions by zero */

extern float **routearr; /* External variables ; defined in driver module */
extern int routei;

/* ************************************************** */
   void normalize (xt,n)
/* ************************************************** */
/* Subroutine for converting vectors to be of length 1.0 */
/* ************************************************** */
float *xt;
int n;
{
 int   i;
 float s = 0.0;

 for (i=1;i<=n;i++)
     s = s + (xt[i] * xt[i]);
 s = sqrt(s);
 for (i=1;i<=n;i++)
     xt[i] = xt[i] / (s + TINY); /* Prevent dividing by zero */
}

/* ************************************************** */
   void gradstep(p,n,fret,func,dfunc)
/* ************************************************** */
/* Subroutine for taking the basic steepest descent step  */
/* ************************************************** */
float *p,*fret,(*func)();
int n;
void (*dfunc)();
{
  float *x;
  int i;

  x = vector(n);
  (*dfunc)(p,x);
  for (i=1;i<=n;i++)
      x[i] = (- x[i]);
  normalize(x,n);
  linmin(p,x,n,fret,func); /* Minimize along -g(p) from p */

  free_vector(x);
}

/* ******************************************************** */
   void modgradstep(pp,p,n,fret,func,dfunc)
/* ******************************************************** */
/* Subroutine for taking the modified steepest descent step  */
/* ******************************************************** */
float *pp,*p,*fret,(*func)();
int n;
```

```
void (*dfunc)();
{
  float *x,*r,*rr,f1,f2;
  int i;

  x = vector(n);
  r = vector(n);
  rr = vector(n);

  /* Compute negative gradient at p */
  (*dfunc)(p,x);
  for (i=1;i<=n;i++)
      x[i] = (- x[i]);
  normalize(x,n);
  /* Minimize along -g(p) from p ; call the new minimum point r */
  for (i=1;i<=n;i++)
      r[i] = p[i];
  linmin(r,x,n,&f1,func);

  /* Compute the "smoothing" direction -pp + 0.5(p+r) */
  for (i=1;i<=n;i++)
      x[i] = - pp[i] + (0.5 * (p[i] + r[i]));
  normalize(x,n);
  /* Minimize along that direction from pp ; call the new minimum
     point rr */
  for (i=1;i<=n;i++)
      rr[i] = pp[i];
  linmin(rr,x,n,&f2,func);

  /* Update the previous point */
  for (i=1;i<=n;i++)
       pp[i] = p[i];

  /* Update the current point p to be either r or rr */
  if (f1 < f2) {
     /* p <- r */
     for (i=1;i<=n;i++)
         p[i] = r[i];
     *fret = f1;
  }
  else {
        /* p <- rr */
        for (i=1;i<=n;i++)
            p[i] = rr[i];
        *fret = f2;
  }
  free_vector(x);
  free_vector(r);
  free_vector(rr);

}

/* ********************************************************* */
   void msteepdesc(p,n,fret,func,dfunc)
/* ********************************************************* */
/* Modified steepest descent method in multidimensions by
   Vuorinen and the author of this study.                    */
/* ********************************************************* */
float *p,*fret,(*func)();
void (*dfunc)();
int n;
{
  int i,counter,dummy;
  float *pp,*vec1,fp,fpp,temp;
```

```
  pp = vector(n);
  vec1 = vector(n);

  for (i=1;i<=n;i++)
      pp[i] = p[i]; /* Initialize the previous point */

  counter = 1;
  fp = (*func)(p); /* Current value of func at p */

  /* Route saving begin */
  if (routei <= MAXROUTEPTS) {
     for (i=1;i<=n;i++)
         routearr[routei][i] = p[i]; /* Save this as a routepoint */
     routei++;                       /* Increase the counter      */
  }
  /* Route saving end */

  /* Initial step is a basic steepest descent step */
  gradstep(p,n,fret,func,dfunc);

  for (counter=2;counter<=MAXITER;counter++) {
      dummy = timeout(); /* if too much CPU-time used then halt */

      /* Route saving begin */
      if (routei <= MAXROUTEPTS) {
         for (i=1;i<=n;i++)
             routearr[routei][i] = p[i]; /* Save this as a routepoint */
         routei++;                       /* Increase the counter      */
      }
      /* Route saving end */

      /* Update variables storing the last two values of function */
      fpp = fp;
      fp  = (*func)(p);

      /*  Termination criterion is the smallness of the gradient */
      /*  or the smallness of change in function value           */

      (*dfunc)(p,vec1);
      temp = 0.0;
      for (i=1;i<=n;i++)
          temp = temp + (vec1[i] * vec1[i]);

      if (sqrt(temp) < TOL1) break; /* Exit routine */

      if ((2.0 * fabs(fp - fpp) <= TOL2 * (fabs(fp) + fabs(fpp))) &&
          (counter > 2)) break;     /* Exit routine */

      modgradstep(pp,p,n,fret,func,dfunc); /* Take the modified step */
  }

  free_vector(pp);
  free_vector(vec1);
}
```

K The implementation of the mathematical functions

```
/* ***************** */
/* Module MATHCORR.C */
/* ***************** */

#include <math.h>

#define TINY 1.0E-10
```

```
#define BIG  1.0E+10

/* This module contains implementations of mathematical functions, called
   from func.c which is generated by Mathematica. Because Mathematica
   uses different kind of names for standard functions than Turbo C++, this
   module is used to interpret them correctly during the compilation.
   In addition to standard C mathematical functions, implemetations of
   some non-standard functions e.g. EllipticE,EllipticK,Hypergeometric2F1
   and ArithmeticGeometricMean are included here. */

double ag(a,b)
double a,b;
{
 double c;
 int    i;

 i = 1;
 while ((fabs(a-b) > TINY) && (i < 8)) {
       c = a;
       a = (a + b) / 2;
       b = sqrt(c * b);
       i = i + 1;
 }
 return((a + b) * 0.5);
}

double ellipk(r)
double r;
{
 if (r <= -TINY) {   /* r < 0 */
    return(BIG);
 }
 if ((r > -TINY) && (r < TINY)) {  /* r == 0 */
     return(2.0 * atan(1.0));
 }
 if (r > 1.0 - TINY) {             /* r >= 1 */
     return(BIG);
 }
 return(2.0 * atan(1.0)/ag(1.0,sqrt(1 - (r * r))));   /* 0 < r < 1 */
}


double ellipe(r)
double r;
{
 double a,b,c,sum;
 int i,m;

 if (r <= -TINY) {  /* r < 0 */
    return(BIG);
 }
 if ((r > -TINY) && (r < TINY)) {  /* r == 0 */
     return(2.0 * atan(1.0));
 }
 if ((r > 1.0 - TINY) && (r < 1.0 + TINY)) {   /* r == 1 */
     return(1.0);
 }
 if (r >= 1.0 + TINY) {                        /* r > 1  */
     return(BIG);
 }

 /* 0 < r < 1 */

 a = 1.0;
 b = sqrt(1 - r*r);
```

```
 i = 1;
 sum = 0.0;
 m = 2;
 while (((a * a) - (b * b) > 1E-12) && (i < 20)) {
       c = a;
       a = (a + b)/2;
       b = sqrt(c*b);
       sum = (a*a-b*b)*m+sum;
       m = (2 * m);
       i=i+1;
 }
 return(2.0 * atan(1.0) * (2 - (r*r) - sum)/(a+b));
}

double hypgeof(a,b,c,r)
double a,b,c,r;
{
 double a1 = a;
 double a2 = b;
 double c1 = c;
 double c2 = c;
 double oneterm = 1.0;
 double sum  = 1.0;
 double g = 1.0;
 double mulr,m,d;
 int j = 0;

 if ((r < -1.0 + TINY) || (r > 1.0 - TINY)) {
    return(BIG);
 }
 while (fabs(oneterm) > 1.0E-8) {
       m = (a1 + j) * (a2 + j);
       d = (c1 + j) * (c2 + j);
       g = g * m/d;
       mulr = pow(r,j+1);
       oneterm = (g * mulr);
       sum = sum + oneterm;
       j++;
 }
 return(sum);
}

double bessj0(x)
double x;
{
 double ax,z;
 double xx,y,ans,ans1,ans2;

 if ((ax=fabs(x)) < 8.0) {
    y=x*x;
    ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
            +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
    ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
            +y*(59272.64853+y*(267.8532712+y*1.0))));
    ans=ans1/ans2;
 }
 else {
    z=8.0/ax;
    y=z*z;
    xx=ax-0.785398164;
    ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
    ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            -y*0.934935152e-7)));
```

```
    ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
 }
 return ans;
}

double bessj1(x)
double x;
{
 double ax,z;
 double xx,y,ans,ans1,ans2;

 if ((ax=fabs(x)) < 8.0) {
    y=x*x;
    ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
            +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
    ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
            +y*(99447.43394+y*(376.9991397+y*1.0))));
    ans=ans1/ans2;
 } else {
    z=8.0/ax;
    y=z*z;
    xx=ax-2.356194491;
    ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
    ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
    ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    if (x < 0.0) ans = -ans;
 }
 return ans;
}

double Abs(x)
double x;
{ return fabs(x);
  }

double ArcCos(x)
double x;
{ return acos(x);
  }

double ArcSin(x)
double x;
{ return asin(x);
   }

double ArcTan(x)
double x;
{ return atan(x);
   }

double BesselJ(i,x)
int i;
double x;
{
  switch (i) {
    case 1  : return bessj1(x);
    case -1 : return -bessj1(x);
    default : return bessj0(x);
  }
}

double Ceiling(x)
```

```
double x;
{ return ceil(x);
   }

double Cos(x)
double x;
{ return cos(x);
  }

double Cosh(x)
double x;
{ return cosh(x);
   }

double Cot(x)
double x;
{ return (1.0 / (tan(x)+TINY));
  }

double Floor(x)
double x;
{ return floor(x);
   }

double Log(x)
double x;
{ return log(x);
   }

double Mod(x,y)
double x,y;
{ return fmod(x,y);
   }

double Power(x,y)
double x,y;
{ return pow(x,y);
   }

double Sec(x)
double x;
{ return (1.0 / (cos(x)+TINY));
  }

double Sin(x)
double x;
{ return sin(x);
  }

double Sinh(x)
double x;
{ return sinh(x);
   }

double Sqrt(x)
double x;
{ return sqrt(x);
   }

double Tan(x)
double x;
{ return tan(x);
   }

double Tanh(x)
```

```
double x;
{ return tanh(x);
   }

double ArithmeticGeometricMean(a,b)
double a,b;
{ return ag(a,b);
   }

double EllipticE(r)
double r;
{ return ellipe(sqrt(r));
   }

double EllipticK(r)
double r;
{ return ellipk(sqrt(r));
   }

double Hypergeometric2F1(a,b,c,r)
double a,b,c,r;
{ return hypgeof(a,b,c,r);
   }
```

L 3-D plots and contour plots of functions 20, 21, 22, 23 and 24

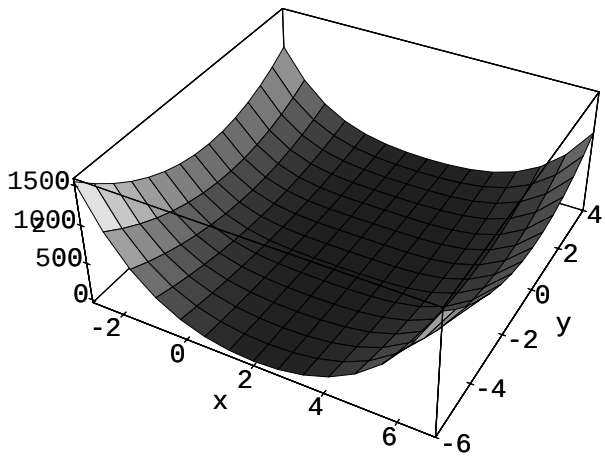

Figure 17: The minimum of function (number 20) $f(x,y) = (x-2)^4+(x-2)^2y^2+(y+1)^2$ is at $\mathbf{x}_* = (2,-1)$

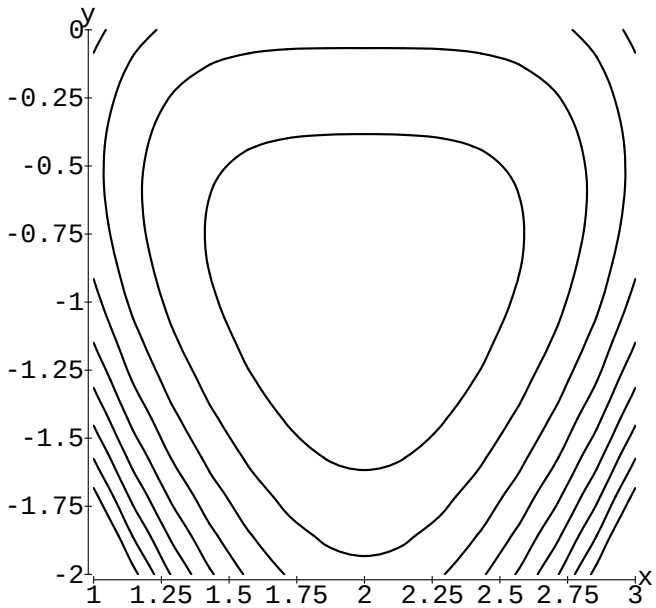

Figure 18: Contour plot of function number 20

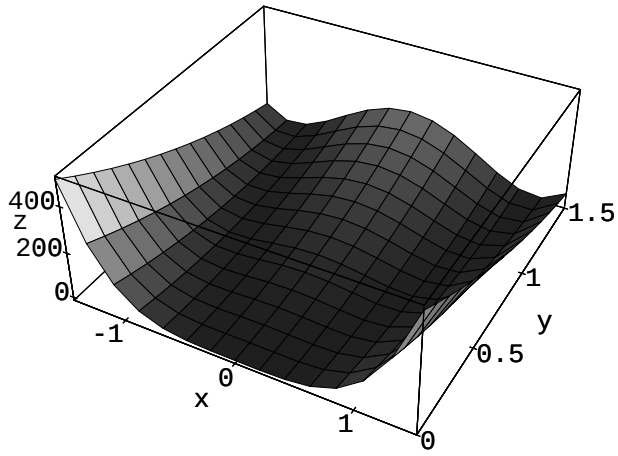

Figure 19: The minimum of the Rosenbrock banana-shaped function (number 21) $f(x, y) = 100(x^2 - y)^2 + (1 - x)^2$ is at $\mathbf{x}_* = (1, 1)$

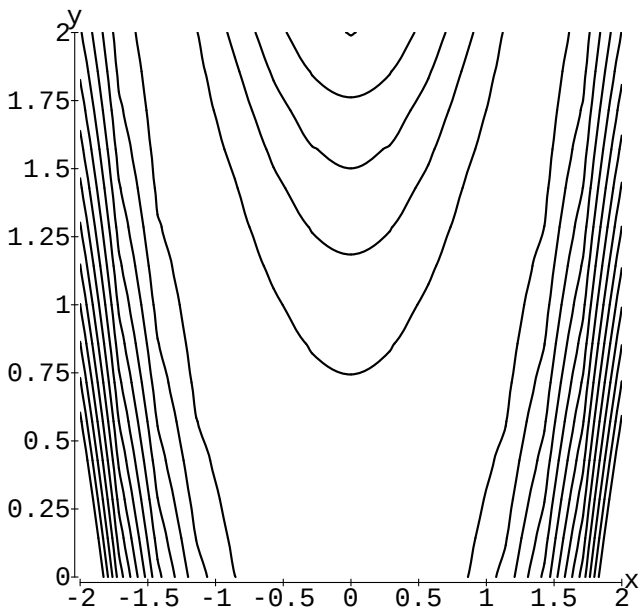

Figure 20: Contour plot of function number 21

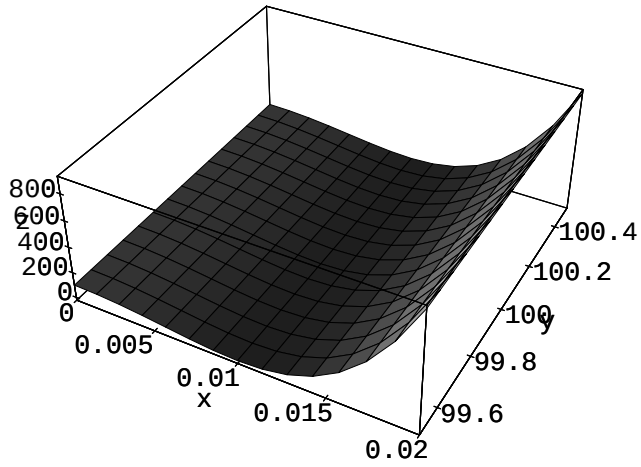

Figure 21: The minimum of function (number 22) $f(x,y) = 100((100x)^2 - \frac{y}{100})^2 + (1 - 100x)^2$ is at $\mathbf{x}_* = (\frac{1}{100}, 100)$

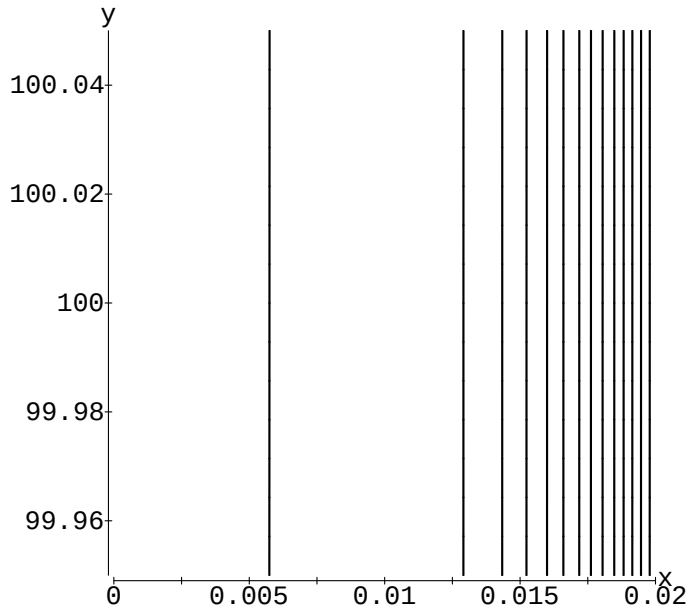

Figure 22: Contour plot of function number 22

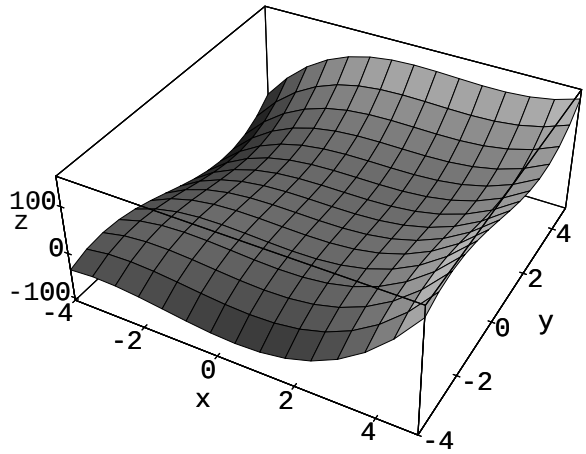

Figure 23: The minimum of function (number 23) $f(x,y) = y^3 - y(x - \frac{1}{\sqrt{3}})^2 + x^3 - x - y$ is at $\mathbf{x}_* = (\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$

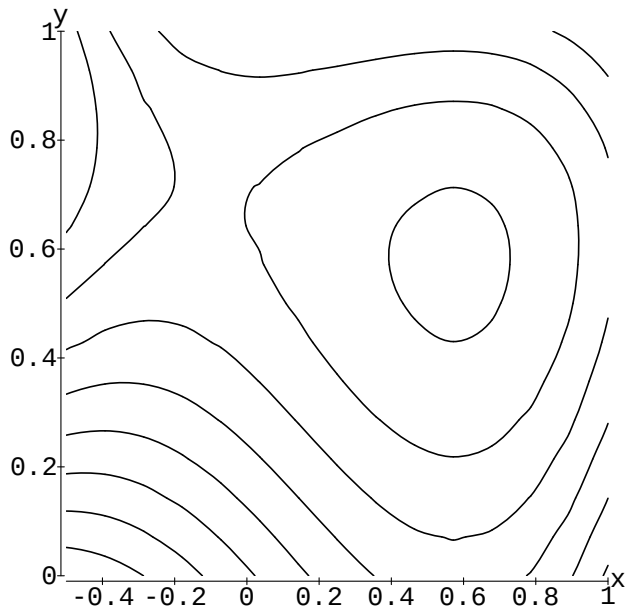

Figure 24: Contour plot of function number 23

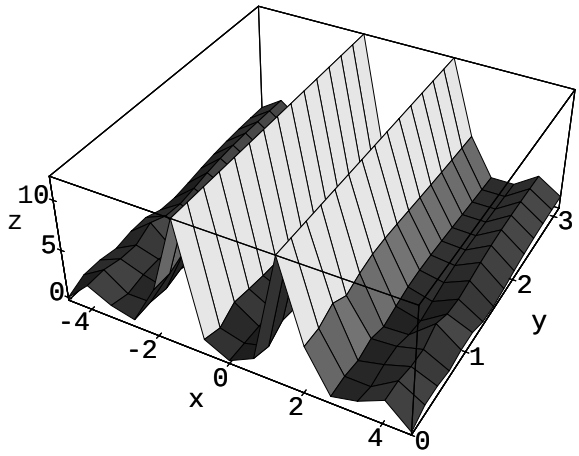

Figure 25: A global minimum of function (number 24) $f(x, y) = \tan^2 x + \sin^2 \frac{x}{y}$ can be found e.g. at $\mathbf{x}_* = (\pi, 1)$

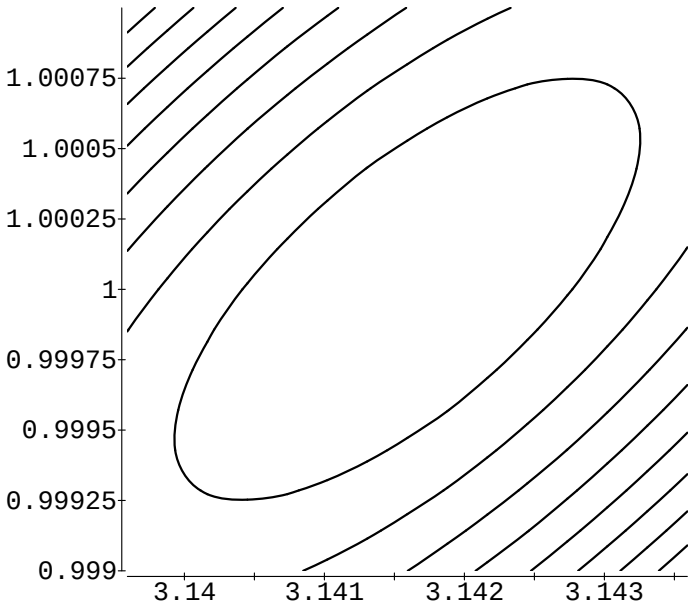

Figure 26: Contour plot of function number 24

References

[Acton 1970] Acton, Forman S. 1970, *Numerical methods that work*. New York : Harper and Row. (pp. 464-467)

[Brent 1973] Brent, Richard P. 1973, *Algorithms for minimization without derivatives*. Englewood Cliffs, N.J. : Prentice-Hall. (p. 78)

[Broyden 1970] Broyden, C.G. 1970, *J.I.M.A.* vol. 6. (pp. 76-90, 222-236)

[Fletcher 1970] Fletcher, R. 1970, *Computer Journal* vol. 13. (pp. 317-322)

[Fletcher-Reeves 1964] Fletcher, R., and Reeves, C.M. 1964, *Computer Journal* vol. 7. (pp. 149-154)

[Kowalik-Osborn 1968] Kowalik, J., and Osborne, M. 1968, *Methods for unconstrained optimization problems*. New York : Elsevier Publishing Co. (pp. 34-39)

[Nash 1990] Nash, J.C. 1990, *Compact numerical methods for computers : linear algebra and function minimization*. Bristol : Hilger. (pp. 148-206)

[Nelder-Mead 1965] Nelder, J.A., and Mead, R. 1965, *Computer Journal* vol. 7. (p. 308)

[Pierre 1969] Pierre, Donald A. 1969, *Optimization theory with applications*. New York : J. Wiley and Sons. (pp. 312-313)

[Press 1986] Press, William H. et al. 1986, *Numerical recipes*. New York : Cambridge University Press. (pp. 274-312)

[Shanno 1970] Shanno, D.F. 1970, *Math. Comp* vol. 24. (p. 647-657)

[Wolfram 1991] Wolfram, S. 1991, *Mathematica : a system for doing mathematics by computer*. Redwood City, California : Addison-Wesley.

Additional literature

Gill, Philip E., Murray, W., and Wright, Margaret H. 1981, *Practical optimization*. London : Academic Press.

Nocedal, J. 1991, *Acta Numerica*. Theory of algorithms for unconstrained optimization. (p. 199-242)